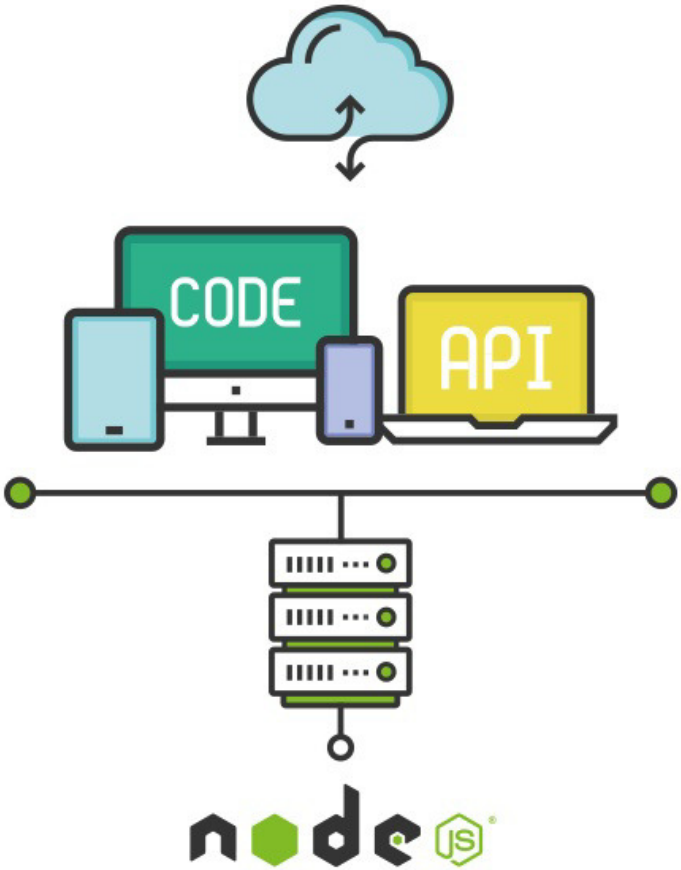
Construindo APIs REST com

Node.js



Casa do Código

CAIO RIBEIRO PEREIRA

ISBN

Impresso e PDF: 978-85-5519-150-3

EPUB: 978-85-5519-151-0

MOBI: 978-85-5519-152-7

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

Primeiramente, quero agradecer a Deus por tudo que fizeste em minha vida! Agradeço também ao meu pai e à minha mãe, pelo amor, força, incentivo e por todo apoio desde o meu início de vida. Obrigado por tudo e, principalmente, por estarem ao meu lado em todos os momentos.

Agradeço à Sra. Charlotte Bento de Carvalho, pelo apoio e incentivo nos meus estudos desde a escola até a minha formatura na faculdade.

Um agradecimento ao meu primo Cláudio Souza. Foi graças a ele que entrei nesse mundo da tecnologia. Ele foi a primeira pessoa a me apresentar o computador, e me aconselhou anos depois a entrar em uma faculdade de TI.

Um agradecimento ao Bruno Alvares da Costa, Leandro Alvares da Costa e Leonardo Pinto, esses caras me apresentaram um mundo novo da área de desenvolvimento de *software*. Foram eles que me influenciaram a escrever um blog, a palestrar em eventos, a participar de comunidades e fóruns e, principalmente, a nunca cair na zona de conforto, a aprender sempre. Foi uma honra trabalhar junto com eles em 2011.

Obrigado ao pessoal da editora Casa do Código, em especial ao Paulo Silveira, Adriano Almeida e Vivian Matsui. Muito obrigado pelo suporte e pela oportunidade!

Obrigado à galera da comunidade NodeBR. Seus *feedbacks* ajudaram a melhorar este livro. Também agradeço a todos os

leitores do blog Underground WebDev (http://udgwebdev.com), afinal, a essência deste livro foi baseada em muitos dos posts do blog.

Por último, obrigado a você, prezado leitor, por adquirir este livro. Espero que ele seja uma ótima referência para você.

COMENTÁRIOS DOS LEITORES

A seguir, veja alguns comentários dos leitores que acompanham meu blog e também gostaram dos meus outros livros, que também foram publicados pela editora Casa do Código.

"Conheci o udgwebdev através do seu livro de Meteor e, desde então, os seus posts vem sempre me surpreendendo na qualidade e simplicidade com que é abordado o conteúdo. Minha maneira de enxergar o JavaScript mudou drasticamente, e a comunidade de Meteor no Brasil só tem a crescer com suas contribuições. Valeu Caio!" — Lucas Nogueira Munhoz — ln.munhoz@gmail.com — http://lucasmunhoz.com

"Mestre no assunto." — Thiago Porto — thiago@waib.com

"Leia a primeira edição do livro NodeJS, sensacional. Ele conduz o leitor a exercitar o conhecimento de forma prática. Parabéns e sucesso!" — Lynneker sales — lynneker@rbmsolutions.com.br

"Tenho os dois livros que você escreveu: Node.js e Meteor, e posso garantir que vou comprar o terceiro. Gosto muito da didática fácil e objetiva que você implementa nos seus livros e no blog. Sempre uso eles como referência no desenvolvimento. Usar ES6 no Front e no Back deve ser lindo demais. Estou ansioso! Nem vi, mas já sei que vou comprar, pois seus artigos e livros nunca decepcionam!" — Nícolas Rossett — nicolas@nvieira.com.br

"Sem dúvida muito bom o livro de Node.js, um conteúdo bem prático, com muitos exemplos e até a construção de um projeto. Isso ajuda muito o leitor, pois a partir do momento que ele põe a mão na

massa, o conteúdo é aprendido de forma mais fácil. Parabéns!" — David Alves — david_end27@hotmail.com

"Apoio demais essa metodologia de aprendizado prático, ainda mais com a construção de um projeto passo-a-passo. Sem dúvidas, pretendo aprender também. Obrigado por disponibilizar esse espaço para a nossa opinião." — Rafael Miguel — raffaelmiguell@gmail.com

SOBRE O AUTOR

Figura 1: Caio Ribeiro Pereira

Um Web Developer com forte experiência no domínio dessa sopa de letrinhas: Node.js, JavaScript, Meteor, Ruby On Rails, Agile, Filosofia Lean, Scrum, XP, Kanban e TDD.

Bacharel em Sistemas de Informação pela Universidade Católica de Santos, blogueiro nos tempos livres, apaixonado por programação, por compartilhar conhecimento, testar novas tecnologias, e assistir filmes e seriados.

Participo das comunidades:

- **NodeBR:** comunidade brasileira de Node.js;
- **MeteorBrasil:** comunidade brasileira de Meteor;
- **DevInSantos:** grupo de desenvolvedores de software em Santos.

Blog: http://udgwebdev.com.

PREFÁCIO

Cenário atual das aplicações web

Atualmente, vivemos em uma fase na qual a maioria dos usuários utilizam diversos tipos de *devices* para se conectarem à internet. Os mais populares são smartphones, tablets e notebooks. Desenvolver sistemas para diversos tipos de devices requer o trabalho de construir *web services*, também conhecidos pelo nome de APIs (*Application Program Interface*).

Basicamente, essas APIs são sistemas back-end que têm o objetivo de trabalhar apenas com dados, de forma centralizada, permitindo que sejam desenvolvidos, separadamente, aplicações clientes que possuem *interfaces* para o usuário final. Essas aplicações clientes geralmente são: *mobile apps*, aplicações *desktop* ou *web apps*.

Desde 2010 até os dias de hoje, o Node.js cada vez mais provou ser uma plataforma excelente na solução de diversos problemas, principalmente para construção de *APIs RESTful*. Sua arquitetura *Single Thread* que realiza I/O não bloqueante rodando em cima do JavaScript – que é uma linguagem muito presente em praticamente todos os browsers atuais – demonstrou uma boa eficiência no processamento de muitas aplicações atuais.

Existem alguns casos de empresas grandes, como por exemplo, LinkedIn e PayPal, que economizaram significativamente gastos com servidores ao migrar alguns de seus projetos para o Node.js.

E uma outra vantagem do uso do Node.js, que cativou muitos

desenvolvedores, foi a sua curva baixa de aprendizado. Afinal, quem já trabalha com desenvolvimento web já possui, pelo menos, um conhecimento básico sobre a linguagem JavaScript.

A quem se destina este livro?

Este livro é destinado aos desenvolvedores web que tenham pelo menos conhecimentos básicos de JavaScript e, principalmente, conheçam bem sobre Orientação a Objetos (OO), arquitetura cliente-servidor e que tenham noções das principais características sobre API RESTful.

Ter domínio desses conceitos, mesmo que seja um conhecimento básico deles, será essencial para que a leitura deste livro seja de fácil entendimento.

Algo bem legal do livro é que todos os códigos utilizarão a mais recente implementação do JavaScript, o EcmaScript 2015 (também conhecido pelos nomes EcmaScript 6 ou ES6).

Como devo estudar?

Ao decorrer da leitura, serão apresentados diversos conceitos e códigos, para que você aprenda na prática toda a parte teórica do livro. Ele o guiará de forma didática no desenvolvimento de dois projetos (uma API e um cliente web), que no final serão integrados para funcionar como um único projeto.

Atenção

É recomendado seguir passo a passo as instruções do livro, para no final você concluir o projeto corretamente.

Sumário

1 Introdução ao Node.js **1**

1.1 O que é Node.js? 1

1.2 Principais características 1

1.3 Por que devo aprender Node.js? 3

2 Setup do ambiente **7**

2.1 Instalação convencional do Node.js 7

2.2 Instalação alternativa via NVM 10

2.3 Test-drive no ambiente 13

3 Gerenciando módulos com NPM **16**

3.1 O que é e o que faz o NPM? 16

3.2 Principais comandos do NPM 17

3.3 Entendendo o package.json 19

3.4 Automatizando tarefas com NPM 22

4 Construindo a API **25**

4.1 Introdução ao Express 25

4.2 Criando o projeto piloto 27

4.3 Implementando um recurso estático 33

4.4 Organizando o carregamento dos módulos 36

5 Trabalhando com banco de dados relacional 43

5.1 Introdução ao SQLite3 e Sequelize 43

5.2 Configurando o Sequelize 45

5.3 Modelando aplicação com Sequelize 48

6 Implementando CRUD dos recursos da API 58

6.1 Organizando rotas das tarefas 58

6.2 Implementando um simples middleware 60

6.3 Listando tarefas via método GET 61

6.4 Cadastrando tarefas via método POST 62

6.5 Consultando uma tarefa via método GET 63

6.6 Atualizando uma tarefa com método PUT 64

6.7 Excluindo uma tarefa com método DELETE 66

6.8 Refactoring no middleware 67

6.9 Implementando rotas para gestão de usuários 68

6.10 Testando rotas com Postman 70

7 Autenticando usuários na API 75

7.1 Introdução ao Passport e JWT 75

7.2 Instalando Passport e JWT na API 78

7.3 Implementando autenticação JWT 79

7.4 Gerando Tokens para usuários autenticados 82

8 Testando a aplicação — Parte 1 90

8.1 Introdução ao Mocha 90

8.2 Configurando ambiente para testes 91

8.3 Testando endpoint de autenticação da API 98

9 Testando a aplicação — Parte 2 **103**

9.1 Testando os endpoints das tarefas 103

9.2 Testando os endpoints de usuário 110

10 Documentando uma API **116**

10.1 Introdução a ferramenta apiDoc 116

10.2 Documentando a geração de tokens 121

10.3 Documentando recurso de gestão de usuários 122

10.4 Documentando recurso de gestão de tarefas 125

11 Preparando o ambiente de produção **130**

11.1 Introdução ao CORS 130

11.2 Habilitando CORS na API 130

11.3 Gerando logs de requisições 132

11.4 Configurando processamento paralelo com módulo cluster 136

11.5 Compactando requisições com GZIP 141

11.6 Configurando SSL para usar HTTPS 143

11.7 Blindando a API com Helmet 144

12 Construindo uma aplicação cliente — Parte 1 **149**

12.1 Setup do ambiente da aplicação 150

12.2 Criando Templates de Signin e Signup 157

12.3 Implementando os componentes de sign in e sign up 160

13 Construindo uma aplicação cliente — Parte 2 **170**

13.1 Templates e componentes para CRUD de tarefas 170

13.2 Componentes para tela de usuário logado 175

13.3 Criando componente de menu da aplicação 178

13.4 Tratando os eventos dos componentes das telas 179

14 Referências bibliográficas 186

Capítulo 1

INTRODUÇÃO AO NODE.JS

1.1 O QUE É NODE.JS?

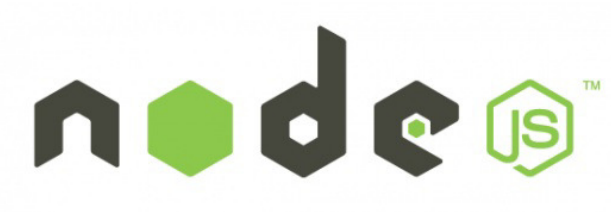


Figura 1.1: Logo do Node.js

O Node.js é uma plataforma altamente escalável e de baixo nível. Nele, você vai programar diretamente com diversos protocolos de rede e internet, ou utilizar bibliotecas que acessam diversos recursos do sistema operacional. Para programar em Node.js, basta dominar a linguagem JavaScript — isso mesmo, JavaScript! E o *runtime* JavaScript utilizado nesta plataforma é o famoso Javascript V8, que é usado também no Google Chrome.

1.2 PRINCIPAIS CARACTERÍSTICAS

Single-thread

Suas aplicações serão *single-thread*, ou seja, cada aplicação terá instância de uma *thread* principal por processo iniciado. Se você está acostumado a trabalhar com programação *multi-thread* — como Java ou .NET —, infelizmente não será possível com Node.js. Porém, existem outras maneiras de se criar um sistema que trabalhe com processamento paralelo.

Por exemplo, você pode utilizar uma biblioteca nativa do Node.js chamada `clusters` , que permite implementar um rede de múltiplos processos de sua aplicação. Nele você pode criar **N-1 processos** de sua aplicação para trabalhar com processamento, enquanto o processo principal se encarrega de balancear a carga entre os demais processos. Se a CPU do seu servidor possui múltiplos núcleos, aplicar essa técnica vai otimizar o seu uso.

Outra maneira é adotar a programação assíncrona, que é um dos principais recursos da linguagem JavaScript. As funções assíncronas no Node.js trabalham com **I/O não bloqueante**, ou seja, caso sua aplicação tenha de ler um imenso arquivo, ela não vai bloquear a CPU, permitindo que ela continue disponível para processar outras tarefas da aplicação realizadas por outros usuários.

> **Observação**
>
> Não se preocupe em entender tudo isso agora, pois veremos mais sobre isso no decorrer do livro.

Event-Loop

Node.js é orientado a eventos. Ele segue a mesma filosofia de orientação de eventos do JavaScript client-side; a única diferença são os tipos de eventos, ou seja, não existem eventos de `click` do mouse, `keyup` do teclado ou qualquer evento de componentes do HTML. Na verdade, trabalhamos com eventos de **I/O**, como por exemplo: o evento `connect` de um banco de dados, um `open` de um arquivo, um `data` de um streaming de dados e muitos outros.

Figura 1.2: Event-Loop do Node.js

O Event-Loop é o agente responsável por escutar e emitir eventos. Na prática, ele é basicamente um **loop infinito** que, a cada iteração, verifica em sua fila de eventos se um determinado evento foi disparado. Quando um evento é disparado, o Event-Loop executa e o envia para a fila de executados. Quando um evento está em execução, nós podemos programar qualquer lógica dentro dele, e isso tudo acontece graças ao mecanismo de `callback` de função do JavaScript.

1.3 POR QUE DEVO APRENDER NODE.JS?

1. **JavaScript everywhere**: praticamente, o Node.js usa JavaScript como linguagem de programação *server-side*. Essa característica permite que você reduza e muito sua curva de aprendizado, afinal, a linguagem é a mesma do JavaScript *client-side*. Seu desafio nesta plataforma será de aprender a fundo como funciona a programação assíncrona para se tirar maior proveito dessa técnica em sua aplicação. Outra vantagem de se trabalhar com JavaScript é que você vai manter um projeto de fácil manutenção — é claro, desde que saiba programar JavaScript de verdade!

   Você terá facilidade em procurar profissionais para seus projetos e gastará menos tempo estudando uma nova linguagem server-side. Uma vantagem técnica do JavaScript comparador com outras linguagens de back-end é que você não vai utilizar mais aqueles frameworks de serialização de objetos JSON (*JavaScript Object Notation*), afinal, o JSON client-side é o mesmo no server-side. Há também casos de aplicações usando banco de dados que persistem objetos JSON, um bom exemplo são os NoSQL MongoDB e CouchDB. Outro detalhe importante é que, atualmente, o Node.js adota diversas funcionalidades da implementação ECMAScript 6, permitindo a codificação de um JavaScript mais elegante e robusto.

2. **Comunidade ativa**: esse é um dos pontos mais fortes do Node.js. Atualmente, existem várias comunidades no mundo inteiro trabalhando muito para esta plataforma, seja divulgando posts e tutoriais, palestrando em eventos, ou principalmente publicando e mantendo novos módulos. Aqui no Brasil, temos três grupos bem ativos:

- **Google**: https://groups.google.com/forum/#!forum/nodebr
- **Facebook**: https://facebook.com/groups/nodejsbrasil
- **Slack**: https://nodebr.slack.com

3. **Ótimos salários**: desenvolvedores Node.js geralmente recebem bons salários. Isso ocorre pelo fato de que infelizmente no Brasil ainda existem poucas empresas adotando essa tecnologia. Isso faz com que empresas que necessitem dela paguem salários na média ou acima da média para manterem esses desenvolvedores em seus projetos. Outro caso interessante são as empresas que contratam estagiários ou programadores juniores que tenham ao menos conhecimentos básicos de JavaScript, com o objetivo de treiná-los para trabalhar com Node.js. Neste caso, não espere um alto salário, e sim um amplo conhecimento preenchendo o seu currículo.

4. **Ready for realtime**: o Node.js ficou popular graças aos seus frameworks de interação realtime entre cliente e servidor. O SockJS e Socket.IO são bons exemplos. Eles são compatíveis com o recente protocolo **WebSockets** e permitem trafegar dados através de uma única conexão bidirecional, tratando todas as mensagens por meio de eventos JavaScript.

5. **Big players**: LinkedIn, Wallmart, Groupon, Microsoft, Netflix, Uber e Paypal são algumas das grandes empresas usando Node.js, e existe muito mais!

Conclusão

Até aqui, foi explicado toda teoria, conceitos e vantagens

principais sobre por que usar o Node.js. Nos próximos capítulos, vamos partir para prática com uma única condição! Abra sua mente para o novo e leia este livro com total empolgação para que você o aproveite ao máximo.

Capítulo 2

SETUP DO AMBIENTE

Neste capítulo, explicarei como instalar o Node.js nos principais sistemas operacionais (Windows, Linux, MacOSX). Porém, no decorrer do livro, os exemplos serão apresentados utilizando um MacOSX.

Apesar de existirem algumas pequenas diferenças de código entre esses sistemas operacionais, fique tranquilo em relação a esse problema, pois os exemplos que serão aplicados neste livro são compatíveis com essas plataformas.

2.1 INSTALAÇÃO CONVENCIONAL DO NODE.JS

Para configurar um ambiente Node.js, independente de qual sistema operacional, as dicas serão praticamente parecidas. Somente alguns procedimentos serão diferentes para cada sistema, principalmente para o Linux, mas não será nada grave.

node
HOME | ABOUT | DOWNLOADS | DOCS | FOUNDATION | GET INVOLVED | SECURITY | NEWS
Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.
Important security notification regarding npm
Download for OS X (x64)
v4.4.3 LTS
Recommended For Most Users
v5.11.0 Stable
Latest Features
Other Downloads | Changelog | API Docs
Other Downloads | Changelog | API Docs
Or have a look at the LTS schedule.
EUROPE
Amsterdam, Netherlands
September 15–16, 2016
node js Interactive
NORTH AMERICA
Austin, Texas
November 29–30, 2016
LINUX FOUNDATION COLLABORATIVE PROJECTS
Report Node.js issue | Report website issue | Get Help
© 2016 Node.js Foundation. All Rights Reserved. Portions of this site originally © 2016 Joyent.

Figura 2.1: Homepage do Node.js

O primeiro passo é acessar seu site oficial do Node.js: http://nodejs.org.

Em seguida, clique no botão *Install* para baixar automaticamente a última versão compatível com seu sistema operacional Windows ou MacOSX. Caso você use Linux, recomendo que leia em detalhes a Wiki do repositório Node.js, em https://github.com/nodejs/node/wiki/Installing-and-Building-Node.js.

Nessa Wiki, é explicado como instalar de forma compilada para qualquer distribuição Linux.

Após fazer o download do Node.js, instale-o normalmente. No caso do Windows e MacOSX, basta clicar no famoso botão *Next* inúmeras vezes até concluir a instalação, pois não há nenhuma configuração específica para ajustar.

Para testar se tudo esta rodando corretamente, abra o terminal (para quem usa Linux ou MacOSX), ou prompt de comandos (se possível utilize o Power Shell) do Windows, e digite o seguinte comando:

```
node -v && npm -v
```

Veja as respectivas versões do Node.js e NPM que foram instaladas. A última versão estável que será utilizada neste livro é o **Node v5.11.0** junto com **NPM 3.8.6**.

Sobre o merge do io.js com Node.js

Desde o dia 8 de setembro de 2015, o Node.js passou da versão **0.12.x** para **4.0.0**. Isso ocorreu devido a um *merge* de uma variação do Node.js chamada **io.js**.

O io.js foi um fork realizado e mantido por um grupo da comunidade Node.js de governança aberta. Eles trabalharam muito na inclusão da nova implementação do **ECMAScript 6**, além da implementação de diversas outras melhorias que eram lentamente trabalhadas no Node.js. Praticamente, a evolução do io.js chegou até a versão **3.0.0**, até que ambos os grupos resolveram fundir o io.js de volta para o Node.js, e assim surgiu a nova versão **4.0.0**. Esse merge não só deu um imenso upgrade nesta plataforma como também tornou-a mais estável e confiável para adoção em projetos de grande porte.

Neste livro, vamos usar a versão **5.11.0**, assim como serão implementados diversos códigos utilizando o novo padrão ECMAScript 6.

Para conhecer as principais funcionalidades do novo JavaScript por meio de exemplos práticos, acesse esse site: http://es6-features.org.

2.2 INSTALAÇÃO ALTERNATIVA VIA NVM

Atenção

Você não é obrigado a instalar o Node.js via NVM. Nesta seção, estamos apenas explorando uma alternativa de instalação do Node.js via gerenciador de versão. Sinta-se à vontade para pular esta etapa caso você já tenha instalado o Node.js de forma convencional e não sente vontade de gerenciar múltiplas versões do Node.js.

Assim como a linguagem Ruby possui o RVM (*Ruby Version Manager*) para gerenciar múltiplas versões do Ruby em uma mesma máquina, o Node.js também possui um gerenciador, que é conhecido por NVM (*Node Version Manager*).

O NVM é a solução perfeita para você que precisa testar o comportamento de seus projetos em distintas versões do Node.js. Ele também serve para a galera que curte testar versões *unstables* também.

O grande benefício do NVM é que ele é prático, fácil de usar, desinstala uma versão Node.js em um único comando e lhe poupará um bom tempo na hora de instalar o Node.js. Ele é uma boa alternativa, principalmente em sistemas Linux cujos *package manager* nativos estão desatualizados e não viabilizam facilmente a instalação de uma versão recente do Node.js.

Configurando o NVM

Em poucas etapas, você configura o NVM para instalá-lo no

MacOSX ou Linux, basta rodar este comando:

```
curl https://raw.githubusercontent.com/creationix/nvm/v0.31.1
                                                    /install.sh \
| bash
```

Infelizmente, o oficial NVM não esta disponível para Windows, mas existem projetos alternativos criados pela comunidade. São duas alternativas bem semelhantes para o Windows:

- **NVMW:** https://github.com/hakobera/nvmw
- **NVM-Windows:** https://github.com/coreybutler/nvm-windows

Ambos possuem uma interface de comandos muito parecida com o NVM.

Principais comandos do NVM

Como receita de bolo, a seguir apresento uma pequena lista com os principais comandos do NVM que serão essenciais para você gerenciar múltiplas versões do Node.js, ou pelo menos manter seu ambiente atualizado com a última versão:

- `nvm ls` : lista todas as versões instaladas em sua máquina;
- `nvm ls-remote` : lista todas as versões disponíveis para download do site http://nodejs.org/dist;
- `nvm install vX.X.X` : baixa e instala uma versão do Node.js;
- `nvm uninstall vX.X.X` : desinstala uma versão Node.js;
- `nvm use vX.X.X` : escolhe uma versão Node.js existente para ser usada;

- `nvm alias default vX.X.X` : escolhe uma versão existente para ser carregada por padrão no início do sistema operacional;
- `nvm help` : lista todos os comandos do NVM.

> **Atenção**
>
> Nos comandos que foram citados `vX.X.X` , você deve trocar o `vX.X.X` por uma versão Node.js de sua escolha, como por exemplo, `v5.11.0` .

Instalando Node.js via NVM

Para instalar o Node.js via NVM, basta rodar os seguintes comandos:

```
nvm install v5.11.0
nvm use v5.11.0
nvm alias default v5.11.0
```

Após a execução desses comandos, você terá o Node.js pré-carregado ao iniciar o seu sistema operacional.

2.3 TEST-DRIVE NO AMBIENTE

Testando Node.js via REPL

Para testarmos o ambiente, executaremos o nosso primeiro programa de `Hello World` , sem criar arquivo de código. Volte ao terminal ou prompt de comando, e execute o comando: `node` .

Este vai acessar o modo REPL (*Read-Eval-Print-Loop*), que permite executar códigos JavaScript diretamente pela tela preta.

Agora digite o comando: `console.log("Hello World")`. Em seguida, tecle `ENTER` para executá-lo na hora. Se tudo der certo, você verá um resultado parecido com a figura a seguir:

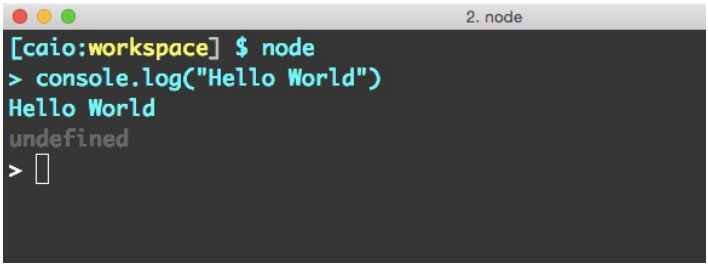


Figura 2.2: Modo REPL do Node.js

Testando Node.js executando código JavaScript

Você também pode fazer um test-drive rodando um arquivo JavaScript contendo o mesmo código anterior. Para isso, crie o arquivo `hello.js` e inclua o seguinte código:

```
console.log("Hello World");
```

Para executá-lo, basta acessar o diretório desse arquivo via terminal, e rodar o comando `node hello.js` para o mesmo resultado do anterior:

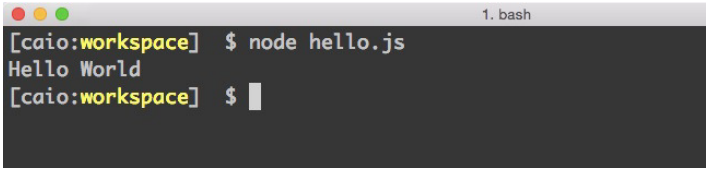


Figura 2.3: Rodando um arquivo JavaScript

Conclusão

Parabéns! Agora, além de ter tudo instalado e funcionando, também aprendeu um novo jeito superlegal sobre como gerenciar múltiplas versões do Node.js. No próximo capítulo, vamos explorar uma outra ferramenta importante do Node.js, o NPM (*Node Package Manager*). Então, continue lendo, pois a brincadeira vai começar!

Capítulo 3

GERENCIANDO MÓDULOS COM NPM

3.1 O QUE É E O QUE FAZ O NPM?

Figura 3.1: Logo do NPM

Assim como o RubyGems do Ruby ou o Maven do Java, o Node.js também possui seu próprio gerenciador de pacotes, que se chama NPM (*Node Package Manager*). Ele se tornou tão popular pela comunidade que, desde a **versão 0.6.X** do Node.js, ele foi integrado no instalador principal do Node.js, tornando-se o gerenciador padrão desta plataforma. Isto simplificou a vida dos desenvolvedores na época, pois fez com que diversos projetos se convergissem para esta ferramenta.

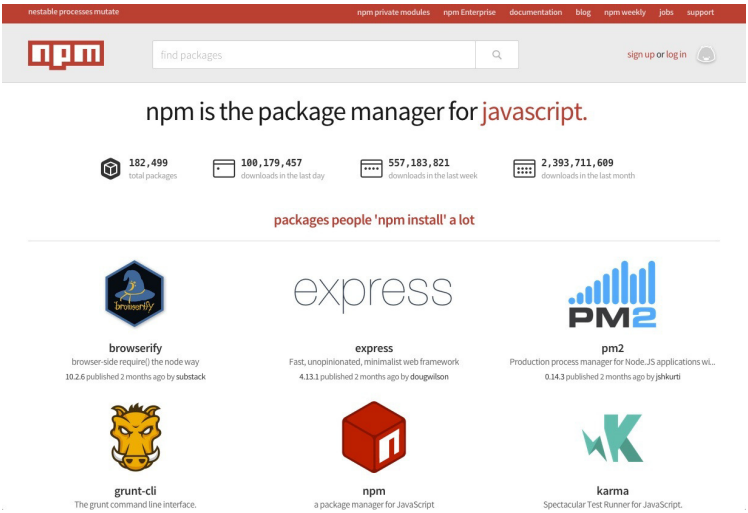


Figura 3.2: Homepage do NPM

Atualmente, o site https://npmjs.org hospeda mais de **250.000 módulos Node.js** criados por terceiros e comunidades. Diariamente são efetuados mais de 180 milhões de downloads e, mensalmente, são cerca de três bilhões de downloads de diversos módulos.

3.2 PRINCIPAIS COMANDOS DO NPM

Utilizar o NPM é muito fácil. Suas utilidades vão além de um simples gerenciador de dependência, pois ele permite também que você crie comandos de automatização de tarefas para seus projetos que são declarados, por meio do arquivo `package.json`. A seguir, veja os principais comandos e seus respectivos significados:

- `npm init`: exibe um miniquestionário para auxiliar na criação e descrição do `package.json` do seu projeto;

- `npm install nome_do_módulo` : instala um módulo no projeto;
- `npm install -g nome_do_módulo` : instala um módulo global;
- `npm install nome_do_módulo --save` : instala o módulo e adiciona-o no arquivo `package.json` , dentro do atributo `"dependencies"` ;
- `npm install nome_do_módulo --save-dev` : instala o módulo e adiciona-o no arquivo `package.json` , dentro do atributo `"devDependencies"` ;
- `npm list` : lista todos os módulos que foram instalados no projeto;
- `npm list -g` : lista todos os módulos globais que foram instalados;
- `npm remove nome_do_módulo` : desinstala um módulo do projeto;
- `npm remove -g nome_do_módulo` : desinstala um módulo global;
- `npm remove nome_do_módulo --save` : desinstala um módulo do projeto, removendo também do atributo `"dependencies"` do `package.json` ;
- `npm remove nome_do_módulo --save-dev` : desinstala um módulo do projeto, removendo também do atributo `"devDependencies"` do `package.json` ;
- `npm update nome_do_módulo` : atualiza a versão de um módulo do projeto;
- `npm update -g nome_do_módulo` : atualiza a versão de um módulo global;
- `npm -v` : exibe a versão atual do NPM;
- `npm adduser nome_do_usuário` : cria um usuário no site

https://npmjs.org;

- `npm whoami` : exibe detalhes do seu perfil público NPM do usuário (é necessário criar um usuário com o comando anterior);
- `npm publish` : publica um módulo no https://npmjs.org (é necessário ter uma conta ativa primeiro);
- `npm help` : exibe em detalhes todos os comandos.

3.3 ENTENDENDO O PACKAGE.JSON

Todo projeto Node.js é chamado de módulo. Mas, o que é um módulo? No decorrer da leitura, perceba que falarei muito sobre o termo módulo, biblioteca e framework e, na prática, eles significam a mesma coisa.

O termo módulo surgiu do conceito de que o JavaScript trabalha com uma arquitetura modular. E quando criamos um projeto, ou seja, um módulo, este é acompanhado de um arquivo descritor de módulos, conhecido pelo nome `package.json` .

Este arquivo é essencial para um projeto Node.js. Um `package.json` mal escrito pode causar bugs ou até impedir o funcionamento do seu projeto, pois ele possui alguns atributos chaves, que são compreendidos tanto pelo interpretador do Node.js como pelo comando `npm` .

Para demonstrar na prática, veja a seguir um exemplo de um simples `package.json` , que descreve os principais atributos de um módulo:

```
{
   "name": "meu-primero-node-app",
   "description": "Meu primeiro app Node.js",
```

```
  "author": "User <user@email.com>",
  "version": "1.2.3",
  "private": true,
  "dependencies": {
    "modulo-1": "1.0.0",
    "modulo-2": "~1.0.0",
    "modulo-3": ">=1.0.0"
  },
  "devDependencies": {
    "modulo-4": "*"
  }
}
```

Com esses atributos, você já descreve o mínimo necessário sobre o que será seu módulo. O atributo `name` é o principal. Com ele, você define o nome do projeto, nome pelo qual seu módulo será chamado via função `require('meu-primeiro-node-app')`. Em `description`, descrevemos o que será este módulo. Ele deve ser escrito de forma curta e clara, fornecendo um resumo sobre o que será.

O `author` é um atributo que informa o nome e e-mail do autor. Utilize o formato `Nome <email>` para que sites, como https://npmjs.org, reconheçam corretamente esses dados.

Outro atributo principal é o `version`, com o qual definimos a versão atual deste módulo. É extremamente recomendado que tenha este atributo, para permitir a instalação de um módulo via comando `npm install meu-primeiro-node-app`. O atributo `private` é opcional, ele é apenas um *boolean* que determina se o projeto será código aberto ou privado.

Os módulos no Node.js trabalham com **3 níveis de versionamento**. Por exemplo, a versão `1.2.3` esta dividida nos níveis:

1. *Major*
2. *Minor*
3. *Patch*

Repare que no campo `dependencies` foram incluídos 4 módulos, sendo que cada um utiliza uma forma diferente de versão.

O primeiro, o `modulo-1`, somente será instalado sua versão fixa, a `1.0.0`. Use este tipo de versão para instalar dependências cujas atualizações possam quebrar o projeto pelo simples fato de que certas funcionalidades foram removidas e ainda as utilizamos na aplicação.

O segundo módulo já possui uma certa flexibilidade de atualização. Ele usa o caractere `~`, que permite atualizar um módulo a nível de `patch` (`1.0.x`). Geralmente, essas atualizações são seguras, trazendo apenas melhorias ou correções de bugs.

O `modulo-3` atualiza versões que sejam maior ou igual a `1.0.0` em todos os níveis de versão. Em muitos casos, usar `>=` pode ser perigoso, porque a dependência pode ser atualizada a um nível *major* ou *minor* e, consequentemente, pode conter grandes modificações que podem quebrar a sua aplicação, exigindo que você ou atualize seu projeto para ficar compatível com a nova versão, ou volte a usar a versão anterior para deixar tudo de volta ao normal.

O último, o `modulo-4`, utiliza o caractere `*` (asterisco). Este sempre pegará a última versão do módulo em qualquer nível. Ele também pode causar problemas nas atualizações e tem o mesmo

comportamento do versionamento do `modulo-3`. Geralmente, ele é usado em `devDependencies`, que são dependências focadas para uso em ambiente de desenvolvimento e testes, em que as atualizações neste tipo de ambiente não prejudicam o comportamento da aplicação que já se encontra em ambiente de produção.

Caso você queira ter um controle mais preciso sobre as versões de suas dependências após a instalação das dependências de seu projeto, rode o comando `npm shrinkwrap`. Ele trava as versões de suas dependências dentro do arquivo `npm-shrinkwrap.json`, de modo que você tenha total controle sobre quais versões foram instaladas em seu projeto. Com isso, você visualiza quais versões estão em seu projeto, inclusive quais versões das dependências de suas dependências foram instaladas. Este comando é para uso em ambiente de produção, onde o controle de versões precisa ser mais rigoroso.

3.4 AUTOMATIZANDO TAREFAS COM NPM

Você também pode automatizar tarefas usando o `npm`. Na prática, você apenas cria novos comandos executáveis por meio do `npm run nome_do_comando`. Para declarar esses novos comandos, basta criá-los dentro do atributo `scripts` no `package.json`. Veja um bom exemplo a seguir:

```
{
   "name": "meu-primero-node-app",
   "description": "Meu primeiro app Node.js",
   "author": "User <user@email.com>",
   "version": "1.2.3",
   "private": true,
   "scripts": {
```

```
      "start": "node app.js",
      "clean": "rm -rf node_modules",
      "test": "node test.js"
   },
   "dependencies": {
      "modulo-1": "1.0.0",
      "modulo-2": "~1.0.0",
      "modulo-3": ">=1.0.0"
   },
   "devDependencies": {
      "modulo-4": "*"
   }
}
```

Repare que foram criados 3 scripts: `start`, `clean` e `test`. Estes agora são executáveis via seus respectivos comandos: `npm run start`, `npm run clean` e `npm run test`. Como *shortcut*, somente os scripts `start` e `test` podem ser executados através dos comandos `npm start` e `npm test`. Dentro de scripts, você pode executar tanto os comandos `node`, `npm` quanto qualquer outro comando global existente em seu sistema operacional. O `npm run clean` é um exemplo disso; nele estou executando o comando `rm -rf node_modules` que, na prática, apaga todo conteúdo da pasta `node_modules`.

Conclusão

Parabéns! Se você chegou até aqui, então você aprendeu o essencial do NPM para gestão de dependências e automatização de tarefas. É de extrema importância dominar pelo menos o básico do NPM, pois ele será usado com muita frequência no decorrer deste livro.

Aperte os cintos e se prepare! No próximo capítulo, começaremos o nosso projeto piloto de API RESTful, que será trabalhado no decorrer dos demais. Como *spoiler*, vou contar aqui

um pouco sobre o que será esse projeto.

Nos próximos capítulos, será desenvolvido na prática uma API REST de um simples sistema de gestão de tarefas. Não só uma API será desenvolvida como também vamos criar uma aplicação cliente web que vai consumir os dados desse serviço. Todo esse desenvolvimento será realizado utilizando alguns frameworks populares do Node.js, como o Express para web framework; Sequelize para lidar com banco de dados *SQL-like*; Passport para lidar com autenticação de usuários, e muito mais. Continue lendo!

Capítulo 4

CONSTRUINDO A API

Agora que temos o ambiente Node.js configurado e pronto para uso, vamos explorar na prática a criação de uma simples aplicação do tipo API RESTful. Esse tipo de aplicação está atualmente sendo desenvolvido por muitos projetos, pois ele traz como vantagem a boa prática de criar uma aplicação focada em apenas servir dados para qualquer tipo de aplicação cliente.

Hoje em dia, é muito comum criar aplicações clientes para web e mobile que consomem dados de uma API. Isso faz com que diversos tipos de aplicações cliente consultem um mesmo servidor centralizado e focado a apenas lidar com dados. Além disso, também permite que cada aplicação – seja ela cliente ou servidor – seja trabalhada isoladamente, por equipes distintas.

Inicialmente, vamos construir uma API, porém, no decorrer dos capítulos, também vamos construir uma simples aplicações cliente web para consumir dados da API. Para começar o desenvolvimento da API, usaremos um framework web muito popular, que se chama Express.

4.1 INTRODUÇÃO AO EXPRESS

O Express é um framework web minimalista, que foi

fortemente inspirado pelo framework Sinatra do Ruby. Com ele, você pode criar desde aplicações pequenas até grandes e complexas, sem nenhum problema. Esse framework permite a construção de APIs e também a criação de aplicações web.

Seu foco é trabalhar com perfeição manipulando `views`, `routes` e `controllers`, ficando à sua escolha trabalhar com `models` e usar qualquer framework de persistência sem gerar nenhum conflito ou incompatibilidade com Express. Isso é uma grande vantagem, pois existem muitos módulos do tipo ODM (*Object Data Mapper*) e também ORM (*Object Relational Mapping*) disponíveis para o Node.js. Você pode usar qualquer um junto com o Express sem a necessidade de configurar alguma integração entre ambos, só precisará carregar os módulos de persistência dentro dos `controllers` ou `routes` de sua aplicação.

O Express permite que o desenvolvimento organize de forma livre os códigos de uma aplicação, ou seja, não existem convenções rígidas neste framework, cada convenção pode ser criada e aplicada. Isso torna-o flexível para adoção tanto em aplicações pequenas quanto aplicações grandes, pois nem sempre é necessário aplicar diversas convenções de organização no projeto para uma pequena aplicação.

Você também pode replicar convenções de outros frameworks. O Ruby On Rails é um exemplo de framework repleto de convenções que vale a pena replicar algumas de suas características. Essa liberdade força o desenvolvedor entender a fundo como funciona cada estrutura da aplicação.

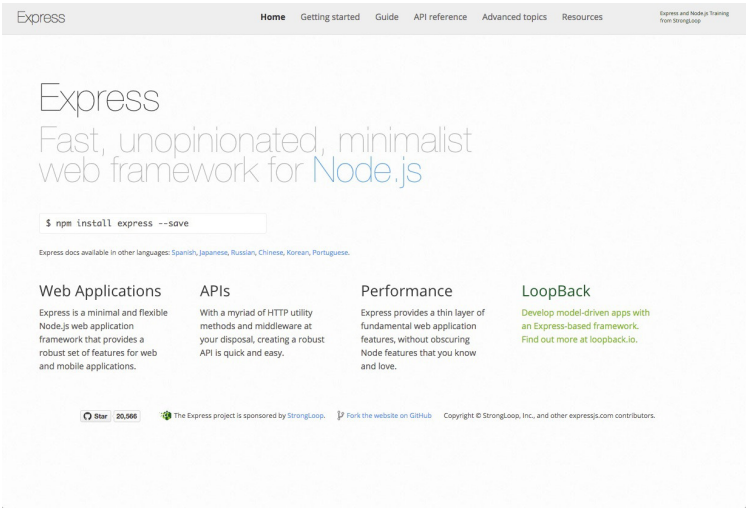

Express
Home
Getting started
Guide
API reference
Advanced topics
Resources
Express and Node.js Training from StrongLoop
Express
Fast, unopinionated, minimalist web framework for Node.js
$ npm install express --save
Express docs available in other languages: Spanish, Japanese, Russian, Chinese, Korean, Portuguese.
Web Applications
Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.
APIs
With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy.
Performance
Express provides a thin layer of fundamental web application features, without obscuring Node features that you know and love.
LoopBack
Develop model-driven apps with an Express-based framework. Find out more at loopback.io.
Star 20,566
The Express project is sponsored by StrongLoop.
Fork the website on GitHub
Copyright © StrongLoop, Inc., and other expressjs.com contributors.

Figura 4.1: Site do Express

Em resumo, veja a seguir uma lista das principais características do Express:

- Routing robusto;
- Facilmente integrável com os principais Template Engines;
- Código minimalista;
- Trabalha com conceito de middlewares;
- Possui uma grande lista de middlewares 3rd-party;
- Content Negotiation;
- Adota padrões e boas práticas de serviços REST.

4.2 CRIANDO O PROJETO PILOTO

Que tal criarmos um projeto na prática? A partir deste capítulo, vamos explorar alguns conceitos de como criar uma API REST utilizando alguns frameworks do Node.js.

Para começar, criaremos um projeto do zero. A nossa aplicação será um simples gerenciador de tarefas, que será dividido em 2 projetos: o primeiro será uma API servidora dos dados, e o segundo será um cliente web consumidora do primeiro.

O nosso projeto terá o nome **NTask** (*Node Task*) e terá as seguintes funcionalidades:

- Listagem de tarefas do usuário;
- Consulta, cadastro, exclusão e alteração de uma tarefa do usuário;
- Consulta, cadastro e exclusão de um usuário;
- Autenticação de usuário;
- Página de documentação da API.

Código-fonte do projeto piloto

Caso você esteja curioso e gostaria de já ir visualizando o código-fonte final dos projetos que serão explorados neste livro, você pode acessar um desses dois links:

- NTask-API: https://github.com/caio-ribeiro-pereira/ntask-api
- NTask-Web: https://github.com/caio-ribeiro-pereira/ntask-web

Nesta aplicação, vamos explorar os principais recursos para se criar uma API REST no Node.js e, no decorrer dos capítulos, vamos incluir novos frameworks importantes para auxiliar no desenvolvimento.

Para começar, vamos criar o primeiro projeto com o nome `ntask-api`, rodando o seguintes comandos:

```
mkdir ntask-api
cd ntask-api
npm init
```

Responda o questionário do comando `npm init` semelhante a este resultado:

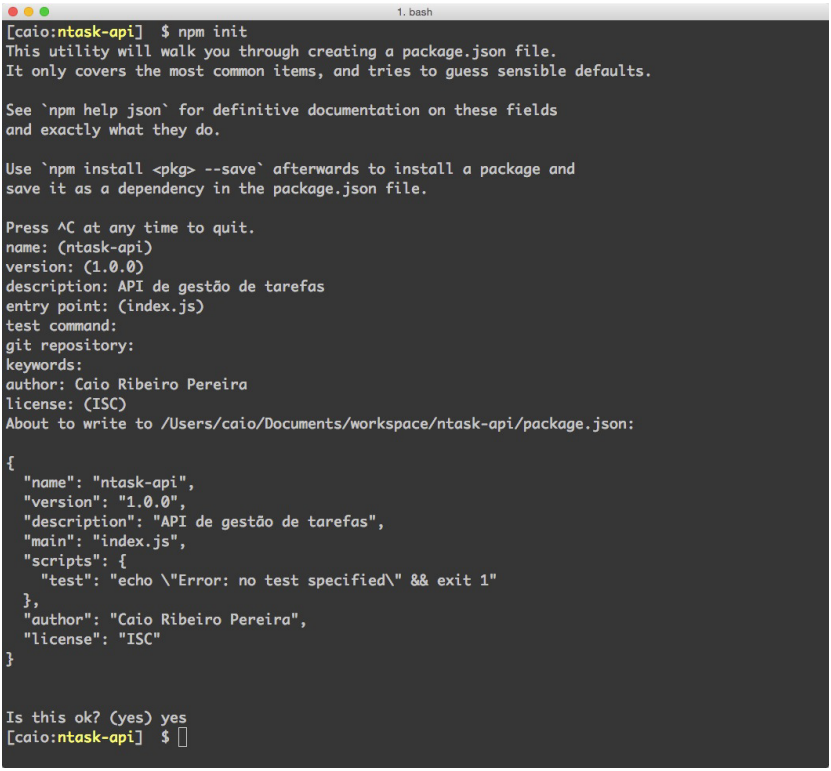

1. bash
[caio:ntask-api] $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
See `npm help json` for definitive documentation on these fields
and exactly what they do.
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.
Press ^C at any time to quit.
name: (ntask-api)
version: (1.0.0)
description: API de gestão de tarefas
entry point: (index.js)
test command:
git repository:
keywords:
author: Caio Ribeiro Pereira
license: (ISC)
About to write to /Users/caio/Documents/workspace/ntask-api/package.json:
{
"name": "ntask-api",
"version": "1.0.0",
"description": "API de gestão de tarefas",
"main": "index.js",
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1"
},
"author": "Caio Ribeiro Pereira",
"license": "ISC"
}
Is this ok? (yes) yes
[caio:ntask-api] $

Figura 4.2: Descrevendo o projeto com npm init

No final, será gerado o arquivo `package.json` com os seguintes campos:

```
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "API de gestão de tarefas",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Caio Ribeiro Pereira",
  "license": "ISC"
}
```

A versão atual do Node.js não tem 100% de suporte ao ES6, porém podemos usar um módulo que emula alguns recursos interessantes para deixar nossa aplicação com códigos lindos do ES6/7. Para isso, vamos instalar o Babel que é um *Transpiler* ES6/7 de código JavaScript compatível, ou seja, vamos implementar códigos ES6/7 e ele vai se encarregar de converter para código JavaScript ES5, caso o Node.js não reconheça nativamente as novas funcionalidades.

> O Babel também é compatível para JavaScript client-side, tanto que nos últimos capítulos deste livro vamos usá-lo para construir uma aplicação web cliente usando ES6 também.
>
> Para conhecer melhor esse projeto, acesse seu site oficial: https://babeljs.io.

Para utilizarmos ao máximo dessas novas funcionalidades do novo JavaScript ES6/7, vamos instalar os módulos `babel-cli` e `babel-preset-es2015` em nosso projeto. Então, execute o comando:

```
npm install babel-cli@6.5.1 babel-preset-es2015@6.5.0 --save
```

Agora precisamos linkar o plugin `babel-preset-es2015` para que o babel reconheça códigos do ES6, para fazer isso, basta criar na raíz do projeto o arquivo `.babelrc` com o seguinte código:

```
{
  "presets": ["es2015"]
}
```

Em seguida, vamos dar uma turbinada no `package.json`. Primeiro, vamos remover os campos `license` e `scripts.test`. Por enquanto, eles não serão usados em nosso projeto.

Depois, incluiremos o campo `scripts.start` para habilitar o comando `npm start`, que será responsável por iniciar nossa API. Esse comando vai compilar o código ES6/7 e iniciar o sistema, tudo isso por meio do comando `babel-node index.js`. Veja a seguir como fica o nosso `package.json`:

```
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "API de gestão de tarefas",
  "main": "index.js",
  "scripts": {
    "start": "babel-node index.js"
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0"
  }
}
```

Agora temos uma descrição mínima do nosso projeto, tudo isso, no arquivo `package.json`. Para começar, vamos instalar o

framework `express`, rodando o seguinte comando:

```
npm install express@4.13.4 --save
```

Com Express instalado, criaremos nosso primeiro código da API. Esse código simplesmente vai carregar o módulo express, criar um simples endpoint `GET /` via função `app.get("/")`, e vai iniciar o servidor na porta `3000` por meio da função `app.listen`. Para isso, crie o arquivo principal `index.js`, implementando este código:

```
import express from "express";
const PORT = 3000;

const app = express();

app.get("/", (req, res) => res.json({status: "NTask API"}));

app.listen(PORT, () => console.log(`NTask API - porta ${PORT}`));
```

Para testar esse código inicial e, principalmente, validar se o esboço da nossa API está funcionando, inicie o servidor com o comando:

```
npm start
```

Sua aplicação deverá apresentar a seguinte mensagem no terminal:

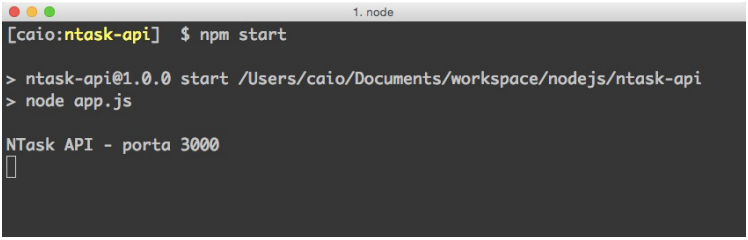


Figura 4.3: Iniciando API pelo terminal

Essas mensagens são referentes ao start da API. Em seguida, abra um browser e acesse o endereço: http://localhost:3000.

Se nenhum problema acontecer, será exibida uma resposta em formato JSON, semelhante a esta figura:

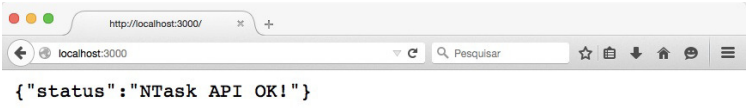


Figura 4.4: JSON de status da API

4.3 IMPLEMENTANDO UM RECURSO ESTÁTICO

O padrão de desenvolvimento de uma API REST trabalha em cima do conceito de criação e manipulação de recursos. Esses recursos basicamente são entidades da aplicação que são utilizadas para consultas, cadastros, atualização e exclusão de dados, ou seja, tudo é baseado em manipular os dados de um recurso.

Por exemplo, a nossa aplicação terá como recurso principal a entidade `tarefas`, que será acessado pelo endpoint `/tasks`. Esta entidade terá alguns dados que descreverão que tipo de informações serão mantidas nesse recurso. Esse conceito segue uma filosofia muito parecida com a modelagem de dados, a única diferença é que os recursos de API REST abstraem a origem dos dados. Ou seja, um recurso pode retornar dados de diferentes fontes, tais como banco de dados, dados estáticos e dados de sistemas externos.

Uma API tem como objetivo tratar e unificar esses dados para, no final, construir e apresentar um recurso. Inicialmente, vamos trabalhar com dados estáticos, porém no decorrer do livro vamos fazer alguns *refactorings* para adotar um banco de dados.

Por enquanto, os dados estáticos serão implementados apenas para moldarmos os endpoints da nossa aplicação. Apenas para moldar nossa API, vamos incluir a rota via função `app.get("/tasks")`, que retornará apenas um JSON de dados estáticos via função `res.json()`. Veja a seguir como serão essas modificações no arquivo `index.js`:

```
import express from "express";
const PORT = 3000;

const app = express();

app.get("/", (req, res) => res.json({status: "NTask API"}));

app.get("/tasks", (req, res) => {
  res.json({
    tasks: [
      {title: "Fazer compras"},
      {title: "Consertar o pc"},
    ]
  });
});

app.listen(PORT, () => console.log(`NTask API - porta ${PORT}`));
```

Para testar esse novo endpoint no terminal, reinicie a aplicação teclando `CTRL+C` ou `Control+C` (se você estiver utilizando MacOSX) e, em seguida, rode novamente o comando:

```
npm start
```

Atenção

Sempre faça essa rotina para reiniciar a aplicação corretamente.

Agora temos um novo endpoint disponível para acessar por meio do endereço: http://localhost:3000/tasks. Ao acessá-lo, você terá o seguinte resultado:

Figura 4.5: Listando tarefas

Caso você queira que seus resultados retornem um JSON formatado e tabulado de forma amigável, inclua no `index.js` a seguinte configuração: `app.set("json spaces", 4);` . Veja a seguir onde incluir essa função:

```
import express from "express";
const PORT = 3000;

const app = express();

app.set("json spaces", 4);

app.get("/", (req, res) => res.json({status: "NTask API"}));

app.get("/tasks", (req, res) => {
```

```
  res.json({
    tasks: [
      {title: "Fazer compras"},
      {title: "Consertar o pc"},
    ]
  });
});

app.listen(PORT, () => console.log(`NTask API - porta ${PORT}`));
```

Agora, reinicie o servidor e veja um resultado mais elegante:

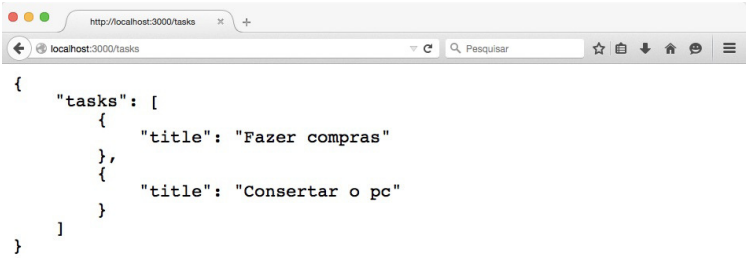


Figura 4.6: Listando tarefas com JSON formatado

4.4 ORGANIZANDO O CARREGAMENTO DOS MÓDULOS

De fato, implementar todos os endpoints no `index.js` não será uma boa prática, principalmente se sua aplicação possuir muitos endpoints. Para isso, vamos organizar os diretórios e carregamento dos códigos de acordo com suas devidas responsabilidades.

Vamos aplicar em nosso projeto o padrão MVR (*Model-View-Router*) para organizar nossa aplicação de forma bem simplificada. Usaremos o módulo `consign`, que permite carregar e injetar

dependências de forma bem simples. Instale-o via comando:

```
npm install consign@0.1.2 --save
```

Com esse módulo instalado, vamos primeiro migrar os endpoints do `index.js`, criando dois novos arquivos de rotas para o novo diretório `routes`. Para isso, crie o código `routes/index.js`:

```
module.exports = app => {
  app.get("/", (req, res) => {
    res.json({status: "NTask API"});
  });
};
```

E também migre o trecho do endpoint `app.get("/tasks")` do arquivo `index.js` para o novo arquivo `routes/tasks.js`:

```
module.exports = app => {
  app.get("/tasks", (req, res) => {
    res.json({
      tasks: [
        {title: "Fazer compras"},
        {title: "Consertar o pc"},
      ]
    });
  });
};
```

Para finalizar essa etapa, edite o `index.js`, para que ele carregue essas rotas por meio do módulo `consign` e inicie o servidor:

```
import express from "express";
import consign from "consign";
const PORT = 3000;

const app = express();

app.set("json spaces", 4);
```

```
consign()
  .include("routes")
  .into(app);

app.listen(PORT, () => console.log(`NTask API - porta ${PORT}`));
```

Pronto! Acabamos de organizar o carregamento dos endpoints da nossa API. Repare que, neste momento, estamos focando apenas em trabalhar com o **VR** (*view e router*) do padrão **MVR**. Em uma API, os resultados em formato JSON são considerados como *views*. A próxima etapa é organizar os *models*.

Para isso, crie o diretório `models` e, voltando no `index.js`, altere os parâmetros da função `consign()` para que primeiro seja carregado os `models` e, depois, os `routes`, utilizando a função `consign().include("models").then("routes")` do mesmo objeto. Para ficar mais clara essa modificação, veja a seguir como deve ficar o carregamento desses módulos:

```
import express from "express";
import consign from "consign";
const PORT = 3000;

const app = express();

app.set("json spaces", 4);

consign()
  .include("models")
  .then("routes")
  .into(app);

app.listen(PORT, () => console.log(`NTask API - porta ${PORT}`));
```

Neste exato momento, a função `consign()` não vai carregar nenhum modelo, inclusive o diretório `models` não possui nenhum código ainda. Para preencher essa lacuna, vamos criar temporariamente um modelo com dados estáticos para

finalizarmos essa primeira etapa, que é organizar o carregamento dos módulos internos.

Para isso, crie o arquivo `models/tasks.js` e implemente este código:

```
module.exports = app => {
  return {
    findAll: (params, callback) => {
      return callback([
        {title: "Fazer compras"},
        {title: "Consertar o pc"},
      ]);
    }
  };
};
```

Esse modelo inicialmente terá apenas a função `Tasks.findAll()`, que receberá dois argumentos como parâmetro: `params` e `callback`. A variável `params` não será utilizada no momento, mas ela servirá de base para enviar alguns filtros de pesquisa SQL, algo que será abordado em detalhes nos próximos capítulos. Já o segundo argumento é função de callback que retorna de forma assíncrona um array estático das tarefas.

Para chamá-lo dentro de `routes/tasks.js`, você terá de carregar esse modelo por meio da varíavel `app`. Afinal, os módulos dos diretórios inseridos na função `consign()` injetam suas lógicas dentro dessa variável principal da aplicação. Para ver como funciona na prática, edite o `routes/tasks.js` da seguinte maneira:

```
module.exports = app => {
  const Tasks = app.models.tasks;
  app.get("/tasks", (req, res) => {
    Tasks.findAll({}, (tasks) => {
      res.json({tasks: tasks});
```

```
    });
  });
};
```

Repare que a função `Tasks.findAll()` possui no primeiro parâmetro um objeto vazio `{}`. Este é o valor da variável `params`, em que, neste caso, não houve necessidade de incluir parâmetros para filtros na listagem das tarefas.

O callback dessa função retorna em seu parâmetro a variável `tasks` que foi criada com valores estáticos dentro do modelo `models/tasks.js`. Então, no momento, temos a total certeza de que `tasks` retornará um array estático com as duas tarefas que foram criadas.

Para finalizar essas modificações, vamos criar um arquivo que carregará toda a lógica dos middlewares e configurações específicas do Express. Atualmente, temos apenas uma simples configuração de formatação JSON, que ocorre via função `app.set("json spaces", 4)`. Vamos incluir mais uma configuração que será a porta do servidor chamando a função `app.set("port", 3000)`.

No decorrer do livro, exploraremos novos middlewares e configurações para o nosso servidor. Logo, já recomendo desde agora a preparar a casa para receber novas visitas!

Crie o arquivo `libs/middlewares.js` seguindo esse código:

```
module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
};
```

Para simplificar, vamos criar o arquivo `libs/boot.js`, que será responsável por iniciar o servidor através da função

`app.listen()`. Nele, vamos remover a constante `PORT` para usar a função `app.get("port")`:

```
module.exports = app => {
  app.listen(app.get("port"), () => {
    console.log(`NTask API - porta ${app.get("port")}`);
  });
}
```

Para finalizar, vamos carregar por último o `libs/boot.js` dentro da estrutura do módulo `consign`. Edite novamente o `index.js` com as seguintes modificações:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("models")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);
```

Para testar essas novas alterações, reinicie seu servidor e acesse novamente o endpoint: http://localhost:3000/tasks.

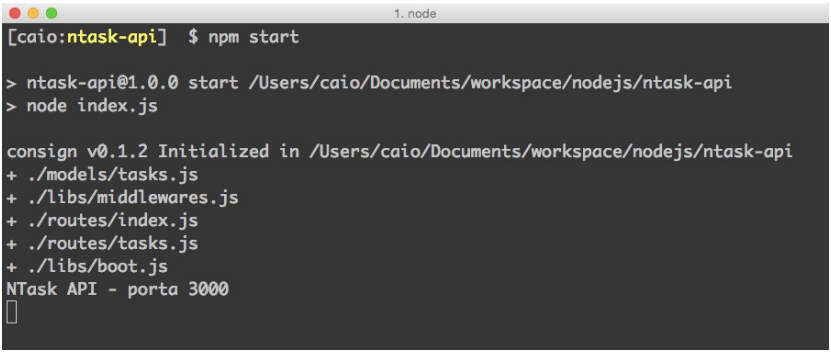


Figura 4.7: Listando modulos carregados

Para ter certeza de que tudo está funcionando corretamente, nenhum erro deve ocorrer e todos os dados das tarefas devem ser exibidos normalmente.

Conclusão

Parabéns, *the mission is complete!* No próximo capítulo, vamos incrementar nosso projeto implementando mais funcionalidades para gerenciar as tarefas de forma dinâmica e usar um banco de dados através do framework Sequelize.

Capítulo 5

TRABALHANDO COM BANCO DE DADOS RELACIONAL

5.1 INTRODUÇÃO AO SQLITE3 E SEQUELIZE

No capítulo anterior, criamos uma estrutura inicial de rotas para a nossa API realizar uma simples listagem de tarefas utilizando um modelo de dados estáticos. Isso foi o suficiente para explorar alguns conceitos básicos de estruturação de nossa aplicação.

Agora, vamos trabalhar mais a fundo na utilização de um banco de dados relacional, que será necessário para implementar uma gestão dinâmica dos dados de nossa aplicação. Para simplificar nossos exemplos, vamos usar o banco SQLite3. Ele é pré-instalado nos sistemas operacionais Linux, Unix e MacOSX, então não há necessidade de configurá-lo. Entretanto, caso você utilize o Windows, você pode facilmente instalá-lo seguindo as instruções desse link: https://www.sqlite.org/download.htm.

Figura 5.1: Logo do SQLite3

O SQLite3 é um banco que armazena todos os dados em um arquivo de extensão `.sqlite` . Ele possui uma interface de linguagem SQL muito semelhante aos demais bancos de dados e está presente não só nos sistemas desktop como também em aplicações mobile.

No Node.js, existem diversos frameworks que trabalham com SQLite3. Em nossa aplicação, vamos usar o módulo Sequelize, que é um módulo completo, e possui uma interface muito bonita e fácil de trabalhar. Nele, será possível manipular dados usando (ou não) comandos SQL, e ele também suporta facilmente os principais bancos de dados SQL, tais como: PostgreSQL, MariaDB, MySQL, SQL Server e SQLite3.

Figura 5.2: Logo do Sequelize

O Sequelize é um framework Node.js do tipo ORM (*Object Relational Mapper*). Suas funções são adotam o padrão *Promises*, que é uma implementação semântica para tratamento de funções

assíncronas presente no ECMAScript 6 do JavaScript.

Atualmente, ele está na versão **3.8.0** e possui funcionalidades para tratamento de transações, modelagem de tabelas, relacionamento de tabelas, replicação de bancos para modo de leitura e muito mais.

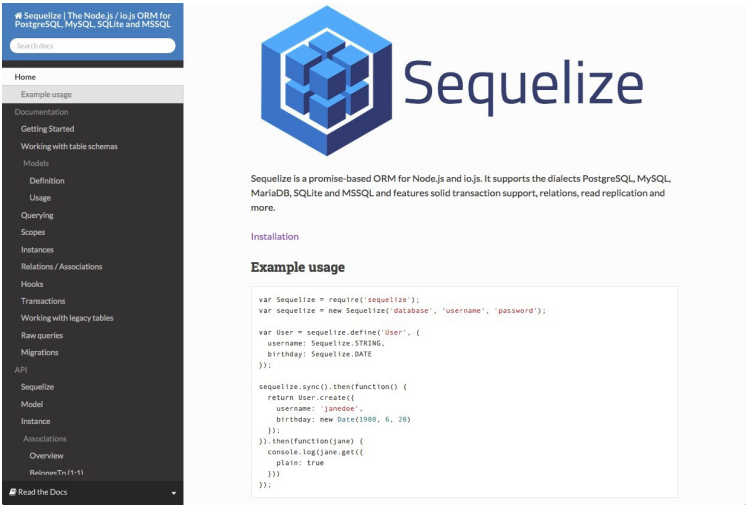


Figura 5.3: Homepage do Sequelize

Seu site oficial com a documentação completa é http://sequelizejs.com.

5.2 CONFIGURANDO O SEQUELIZE

Para começarmos com o Sequelize, basta executar no terminal o seguinte comando:

```
npm install sequelize@3.19.2 sqlite3@3.1.1 --save
```

Com esses dois módulos instalados, já temos em nosso projeto

as dependências necessárias para nossa API se conectar em um banco de dados. Agora, vamos criar um arquivo de configuração de conexão entre o Sequelize com o SQLite3. Para isso, crie o arquivo o `libs/config.js` com os seguintes parâmetros:

- `database` – define o nome da base de dados;
- `username` – informa o nome de usuário de acesso;
- `password` – informa a senha do usuário;
- `params.dialect` – informa qual é o banco de dados a ser usado;
- `params.storage` – é um atributo específico para o SQLite3, sendo que nele é informado o diretório que será gravado o arquivo da base de dados;
- `params.define.underscored` – padroniza o nome dos campos da tabela em minúsculo usando *underscore* no lugar dos espaços em branco.

Veja a seguir como fica esse arquivo:

```
module.exports = {
  database: "ntask",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    define: {
      underscored: true
    }
  }
};
```

Após criar esse simples arquivo de configuração, vamos agora criar o código responsável pela conexão com o banco de dados que usará essas configurações. Esse código de conexão adotará o design pattern Singleton, ou seja, ele vai garantir que seja instanciada

apenas uma vez a conexão do Sequelize. Isso vai permitir carregar inúmeras vezes esse módulo realizando apenas uma única conexão com o banco de dados.

Para isso, crie o código `db.js` da seguinte maneira:

```
import Sequelize from "sequelize";
const config = require("./libs/config.js");
let sequelize = null;

module.exports = () => {
  if (!sequelize) {
    sequelize = new Sequelize(
      config.database,
      config.username,
      config.password,
      config.params
    );
  }
  return sequelize;
};
```

Pronto! Para iniciar esse módulo de conexão, vamos incluí-lo no carregamento de módulos do `consign`. O `db.js` será o primeiro módulo a ser executado (por meio da função `consign().include("db.js");`), pois os demais códigos da aplicação usarão essa instância de conexão do Sequelize para manipulação dos dados. Para implementar isso, edite o `index.js`:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("db.js")
  .then("models")
  .then("libs/middlewares.js")
  .then("routes")
```

```
  .then("libs/boot.js")
  .into(app);
```

Para finalizar o setup do Sequelize, vamos implementar uma simples função de sincronização, entre o Sequelize com o banco de dados. Essa sincronia realiza, se necessário, alterações nas tabelas do banco de dados, de acordo com o que for configurado nos modelos da aplicação. Para incluir a função `app.db.sync()`, para que sincronize as tabelas do banco de dados com os modelos do Sequelize, vamos editar o `libs/boot.js`, baseando-nos no código a seguir:

```
module.exports = app => {
  app.db.sync().done(() => {
    app.listen(app.get("port"), () => {
      console.log(`NTask API - porta ${app.get("port")}`);
    });
  });
}
```

Para testar essas modificações, reinicie o servidor. Se estiver tudo certo, sua aplicação deverá funcionar do jeito que estava antes, afinal, nenhuma modificação visível foi realizada, apenas algumas adaptações foram implementadas para tornar nossa aplicação conectável a um banco de dados. Na próxima seção, modificaremos toda a modelagem de dados, e isso sim terá um grande impacto nas mudanças.

5.3 MODELANDO APLICAÇÃO COM SEQUELIZE

Até agora, o único modelo de nossa aplicação, o `models/tasks.js`, está retornando dados estáticos via função `Tasks.findAll()`. Isso foi necessário, pois precisávamos, no

capítulo anterior, preparar a estrutura de diretórios e carregamento dos módulos da aplicação.

Nesta seção, vamos explorar as principais funcionalidades do Sequelize para a criação de modelos que representarão as tabelas do nosso banco de dados e os recursos de nossa API.

Nossa aplicação terá apenas dois modelos: `Users` e `Tasks`. O relacionamento entre essas tabelas será de `Users 1-N Tasks`, semelhante a esta figura:

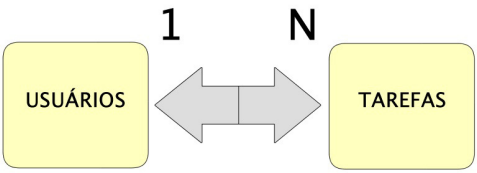


Figura 5.4: Modelagem da base de dados

Para trabalhar com esse tipo de relacionamento, usaremos as funções do Sequelize: `Users.hasMany(Tasks)` (no futuro `models/users.js`) e `Tasks.belongsTo(Users)` (no `models/tasks.js`). Essas associações serão encapsuladas dentro de um atributo chamado `classMethods`, que permite incluir funções estáticas para o modelo.

Em nosso caso, vamos criar a função `associate` dentro de um `classMethods` de cada modelo. Assim poderemos executar essa função de relacionamento de tabelas dentro do `db.js`, que será em breve modificado para atender essa necessidade.

Criando modelo Tasks

Para iniciar essa brincadeira, vamos começar modificando e modelando o arquivo `models/tasks.js` , aplicando as seguintes alterações:

```
module.exports = (sequelize, DataType) => {
  const Tasks = sequelize.define("Tasks", {
    id: {
      type: DataType.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    title: {
      type: DataType.STRING,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    },
    done: {
      type: DataType.BOOLEAN,
      allowNull: false,
      defaultValue: false
    }
  }, {
    classMethods: {
      associate: (models) => {
        Tasks.belongsTo(models.Users);
      }
    }
  });
  return Tasks;
};
```

A função `sequelize.define("Tasks")` é responsável por criar ou alterar uma tabela no banco de dados. Isso ocorre quando o Sequelize faz uma sincronização no *boot* da aplicação. Seu segundo parâmetro é um objeto, e seus atributos representam respectivamente os campos de uma tabela, e seus valores são

subatributos descritores do tipo de dados desses campos.

Neste modelo, o campo `id` é do tipo inteiro (`DataType.INTEGER`). Ele representa uma chave primária (`primaryKey: true`) e seu valor é autoincremental (`autoIncrement: true`) a cada novo registro.

O campo `title` é do tipo *string* (`DataType.STRING`). Nele foi incluído o atributo `allowNull: false` para não permitir valores nulos, e também um campo validador, que verifica se a string não é vazia (`validate.notEmpty: true`). O campo `done` é do tipo *boolean* (`DataType.BOOLEAN`), que não permite valores nulos (`allowNull: false`). Aliás, se não for informado um valor para este campo, ele será por padrão registrado como **false** (`defaultValue: false`).

Por último, temos um terceiro parâmetro que permite incluir funções estáticas encapsulados dentro do atributo `classMethods` . Nele foi criada a função `associate(models)` , que vai permitir realizar uma associação entre os modelos. Neste caso, o relacionamento foi estabelecido por meio da função `Tasks.belongsTo(models.Users)` . Esse é um relacionamento do tipo `Tasks 1-N Users` .

> **Atenção**
>
> Ainda não foi criado o modelo `Users` . Então, se você reiniciar o servidor, um erro acontecerá. Fique tranquilo e continue lendo para finalizarmos a modelagem de nossa aplicação.

Criando modelo Users

Para completar nossa simples modelagem, vamos criar o modelo que vai representar os usuários de nossa aplicação. Crie o arquivo `models/users.js` com as seguintes definições:

```
module.exports = (sequelize, DataType) => {
  const Users = sequelize.define("Users", {
    id: {
      type: DataType.INTEGER,
      primaryKey: true,
      autoIncrement: true
    },
    name: {
      type: DataType.STRING,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    },
    password: {
      type: DataType.STRING,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    },
    email: {
      type: DataType.STRING,
      unique: true,
      allowNull: false,
      validate: {
        notEmpty: true
      }
    }
  }, {
    classMethods: {
      associate: (models) => {
        Users.hasMany(models.Tasks);
      }
    }
  });
  return Users;
```

```
};
```

Dessa vez, a modelagem dos campos da tabela `Users` foi muito semelhante ao modelo `Tasks`. A única diferença foi a inclusão do atributo `unique: true`, dentro do campo `email`, para garantir que não cadastrem e-mails repetidos neste campo.

Após terminar essa etapa de modelagem, vamos agora alterar alguns códigos existentes no projeto, para que eles possam carregar corretamente esses modelos e executar suas respectivas funções de relacionamento entre tabelas. Para começar, vamos modificar no `index.js` o carregamento de alguns módulos.

Em primeiro lugar, vamos ordenar o carregamento dos módulos para que o `libs/config.js` carregue primeiro e, em seguida, o `db.js`. Também vamos remover o carregamento do diretório `models` do `consig`. Veja como o código deve ficar:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("libs/config.js")
  .then("db.js")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);
```

O motivo da exclusão do diretório `models` via módulo `consign` é que implementaremos todo o carregamento dos modelos diretamente pelo arquivo `db.js`, por meio da função `sequelize.import()`. Afinal, se você voltar nos códigos dos modelos, perceberá que surgiram dois novos atributos dentro de

`module.exports = (sequelize, DataType)`. Estes serão magicamente injetados via função `sequelize.import`, que é responsável por carregar e definir os modelos no banco de dados. Praticamente, faremos o seguinte *refactoring* no código `db.js`:

```
import fs from "fs";
import path from "path";
import Sequelize from "sequelize";

let db = null;

module.exports = app => {
  if (!db) {
    const config = app.libs.config;
    const sequelize = new Sequelize(
      config.database,
      config.username,
      config.password,
      config.params
    );
    db = {
      sequelize,
      Sequelize,
      models: {}
    };
    const dir = path.join(__dirname, "models");
    fs.readdirSync(dir).forEach(file => {
      const modelDir = path.join(dir, file);
      const model = sequelize.import(modelDir);
      db.models[model.name] = model;
    });
    Object.keys(db.models).forEach(key => {
      db.models[key].associate(db.models);
    });
  }
  return db;
};
```

Dessa vez, o código ficou um pouco complexo, não é verdade? Porém, sua funcionalidade ficou bem legal! Agora podemos utilizar as configurações de banco de dados através do objeto

`app.libs.config`. Outro detalhe está na execução da função encadeada `fs.readdirSync(dir).forEach(file)`, que basicamente vai retornar um array de strings referente aos nomes de arquivos existentes no diretório `models`. Depois, esse array será iterado, para que dentro de seu escopo de iteração sejam carregados todos os modelos via função `sequelize.import(modelDir)` e, em seguida, inseridos nesse modelo dentro da estrutura `db.models` por meio do trecho `db.models[model.name] = model`.

Após carregar todos os modelos, uma nova iteração ocorre através da função `Object.keys(db.models).forEach(key)`. Ela basicamente executará a função `db.models[key].associate(db.models)` para garantir o relacionamento correto entre os modelos.

Para terminar as adaptações, ainda temos de fazer uma simples alteração no código `libs/boot.js`, mudando a chamada da função `app.db.sync()` para `app.db.sequelize.sync()`:

```
module.exports = app => {
  app.db.sequelize.sync().done(() => {
    app.listen(app.get("port"), () => {
      console.log(`NTask API - porta ${app.get("port")}`);
    });
  });
}
```

Em seguida, edite o `routes/tasks.js` para que ele carregue o modelo corretamente pela chamada `app.db.models.Tasks`, e modifique a função `Tasks.findAll()` para o padrão *promises* do Sequelize. Veja a seguir como fica:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;
```

```
  app.get("/tasks", (req, res) => {
    Tasks.findAll({}).then(tasks => {
      res.json({tasks: tasks});
    });
  });
};
```

Conclusão

Finalmente terminamos essa adaptação do Sequelize em nossa aplicação. Para testar se ela foi corretamente implementada, basta reiniciar o servidor, e você verá no terminal (ou prompt de comandos) uma mensagem semelhante a esta:

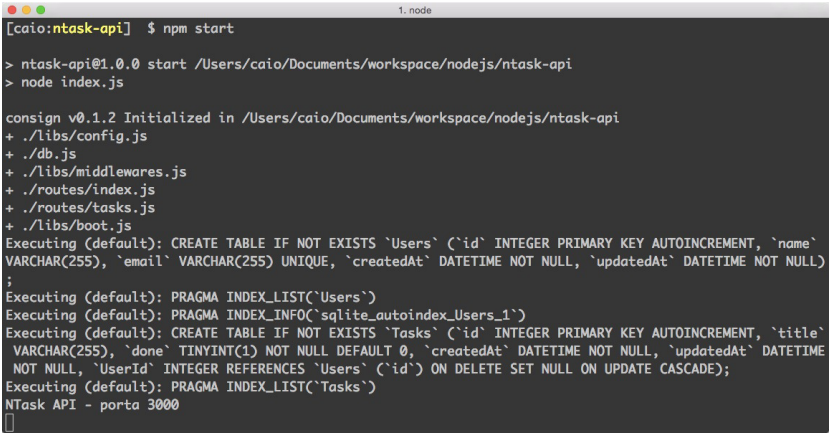

1. node
[caio:ntask-api] $ npm start
> ntask-api@1.0.0 start /Users/caio/Documents/workspace/nodejs/ntask-api
> node index.js
consign v0.1.2 Initialized in /Users/caio/Documents/workspace/nodejs/ntask-api
+ ./libs/config.js
+ ./db.js
+ ./libs/middlewares.js
+ ./routes/index.js
+ ./routes/tasks.js
+ ./libs/boot.js
Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `name` VARCHAR(255), `email` VARCHAR(255) UNIQUE, `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL);
Executing (default): PRAGMA INDEX_LIST(`Users`)
Executing (default): PRAGMA INDEX_INFO(`sqlite_autoindex_Users_1`)
Executing (default): CREATE TABLE IF NOT EXISTS `Tasks` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `title` VARCHAR(255), `done` TINYINT(1) NOT NULL DEFAULT 0, `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, `UserId` INTEGER REFERENCES `Users` (`id`) ON DELETE SET NULL ON UPDATE CASCADE);
Executing (default): PRAGMA INDEX_LIST(`Tasks`)
NTask API - porta 3000

Figura 5.5: Criação das tabelas no banco de dados

Se você acessar o endereço http://localhost:3000/tasks, dessa vez retornará um objeto JSON sem tarefas.

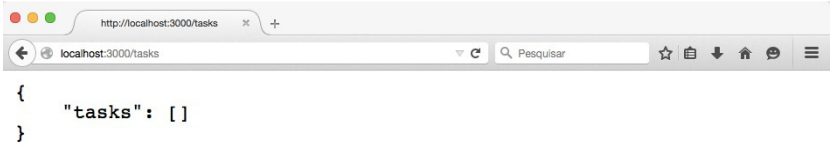


Figura 5.6: Agora, a lista de tarefas está vazia!

Mas não se preocupe, pois no próximo capítulo será implementado os principais endpoints para realizar um CRUD completo. Então, *keep reading*!

CAPÍTULO 6

IMPLEMENTANDO CRUD DOS RECURSOS DA API

Neste capítulo, vamos explorar a fundo o uso de novas funções do Sequelize e também algumas técnicas de organização de rotas e middlewares do Express. Implementaremos praticamente um CRUD (*Create, Read, Update, Delete*) dos modelos `Tasks` e `Users`.

6.1 ORGANIZANDO ROTAS DAS TAREFAS

Para começar esse *refactoring*, vamos explorar os principais métodos do HTTP para CRUD. Neste caso, usaremos as funções `app.route("/tasks")` e `app.route("/tasks/:id")` para definir dois endpoints: `"/tasks"` e `"/tasks/(id_da_task)"`.

Essas funções permitirão, por meio de funções encadeadas, o reúso desses endpoints através das outras funções do Express, que são referentes aos métodos do HTTP. Com isso, poderemos usar as funções:

- `app.all()`: é um middleware que é executado via qualquer método do HTTP;
- `app.get()`: executa o método `GET` do HTTP, ele é usado

para consultas no sistema e, geralmente, retorna algum conjunto de dados de um ou múltiplos recursos;

- `app.post()` : executa o método `POST` do HTTP, ele é semanticamente usado para cadastrar novos dados em um recurso;
- `app.put()` : executa o método `PUT` do HTTP, muito usado para atualizar dados de um recurso da API;
- `app.patch()` : executa o método `PATCH` do HTTP, possui uma semântica parecida com o PUT, porém seu uso é recomendado apenas para atualizar alguns atributos de um recurso, e não todos os dados dele;
- `app.delete()` : executa o método `DELETE` do HTTP, assim como seu nome diz, ele é usado para excluir um determinado recurso da API.

Para entender melhor o uso dessas rotas, vamos editar o `routes/tasks.js` , aplicando as funções necessárias para estruturar um CRUD de tarefas:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;

  app.route("/tasks")
    .all((req, res) => {
      // Middleware de pré-execução das rotas
    })
    .get((req, res) => {
      // "/tasks": Lista tarefas
    })
    .post((req, res) => {
      // "/tasks": Cadastra uma nova tarefa
    });
  app.route("/tasks/:id")
    .all((req, res) => {
      // Middleware de pré-execução das rotas
    })
    .get((req, res) => {
```

```
      // "/tasks/1": Consulta uma tarefa
    })
    .put((req, res) => {
      // "/tasks/1": Atualiza uma tarefa
    })
    .delete((req, res) => {
      // "/tasks/1": Exclui uma tarefa
    });
};
```

Com essa estrutura mínima, já temos um esboço das rotas necessárias para gerenciar tarefas. Agora, podemos implementar suas respectivas lógicas para tratar corretamente cada ação do nosso CRUD de tarefas.

6.2 IMPLEMENTANDO UM SIMPLES MIDDLEWARE

Nos dois endpoints (`/tasks` e `/tasks/id`) teremos de tratar algumas regrinhas no middleware `app.all()` para evitar problemas de acesso no envio do atributo `id` de uma tarefa. Esse tratamento será uma regra bem simples, veja a seguir como deve ficar:

```
app.route("/tasks")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  // Continuação dos routers...
app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  // Continuação dos routers...
```

Praticamente, estamos garantindo a exclusão do atributo `id`

dentro do corpo de uma requisição, ou seja, não será permitido o `req.body.id` nas requisições. Isso porque, nas funções de cada requisição, usaremos o `req.body` como parâmetro das funções do Sequelize, e o atributo `req.body.id` poderá sobrescrever o `id` de uma tarefa, por exemplo, no `update` ou `create` de uma tarefa.

Para finalizar o middleware, avisando-o que deve executar uma função respectiva a um método do HTTP, basta incluir no final do callback a função `next()` para ele avisar ao roteador do Express que ele pode executar a próxima função da rota ou um próximo middleware abaixo.

6.3 LISTANDO TAREFAS VIA MÉTODO GET

Já temos um simples middleware de tratamento para o recurso `/tasks`, agora vamos, por partes, implementar suas funções de CRUD. Para começar, implementaremos a função `app.get()`, que listará dados do modelo `Tasks` do Sequelize, executando a função `Tasks.findAll()`:

```
app.route("/tasks")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findAll({})
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
```

Nesta primeira implementação, vamos listar todas as tarefas do

banco por meio da execução: `Tasks.findAll({})` . Apesar de ser uma má prática listar todos os dados por questões didáticas, vamos deixá-la assim. Mas fique tranquilo que no decorrer dos capítulos serão implementados alguns argumentos nessa função para garantir uma listagem mais específica das tarefas.

O resultado da consulta ocorre por meio da função `then()` e, caso exista algum problema nela, você pode tratar os erros através da função `catch()` . Outro detalhe são os status do HTTP a serem usados, pois uma resposta de sucesso retornará por *default* o status `200 - OK` . Em nossa API, vamos usar o status `412 - Precondition Failed` para retornar os demais erros de validação de campo, ou erros de acesso na base de dados.

SOBRE OS STATUS DO HTTP

Não há uma regra rígida a ser seguida na definição dos status do HTTP, porém é recomendável que se entenda o significado dos principais status para tornar os status de resposta mais semântico.

Para conhecer os demais status do HTTP, recomendo a leitura desse link oficial do site W3: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

6.4 CADASTRANDO TAREFAS VIA MÉTODO POST

Não há muito segredo na implementação do método POST.

Vamos basicamente usar a função `Tasks.create(req.body)` e, em seguida, tratar seus resultados.

O mais interessante do Sequelize é que sua modelagem já faz uma limpeza dos parâmetros que não fazem parte do modelo. Isso é muito bom, pois caso o `req.body` contenha diversos atributos que não foram definidos para o modelo, eles serão descartados na hora da inserção da tarefa. O único problema é a possível existência do atributo `req.body.id`, que poderia adulterar o mecanismo de `id` autoincremental do banco de dados. Entretanto, isso já foi tratado no middleware da função `app.all()`, e o resultado de sucesso retorna o próprio objeto da tarefa criada. A implementação dessa rota fica da seguinte maneira:

```
app.route("/tasks")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findAll({})
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
  .post((req, res) => {
    Tasks.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
```

6.5 CONSULTANDO UMA TAREFA VIA

MÉTODO GET

Já finalizamos as funções do endpoint `/tasks`, agora vamos tratar as do `/tasks/:id`. Para isso, vamos começar com a implementação da função `app.route("/tasks/:id")`, que também terá a mesma lógica de middleware da função `.all()` anterior.

Para finalizar, usaremos a função `Tasks.findOne({where: req.params})`, que executará, por exemplo, `Tasks.findOne({where: {id: "1"}})`. Ela faz uma consulta unitária de tarefas baseada no seu `id` do banco de dados e, caso não exista uma tarefa, vamos responder utilizando o status `404 - Not Found` do HTTP via função `res.sendStatus(404)`, que significa que nada foi encontrado. Veja como fica:

```
app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findOne({where: req.params})
      .then(result => {
        if (result) {
          res.json(result);
        } else {
          res.sendStatus(404);
        }
      })
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
```

6.6 ATUALIZANDO UMA TAREFA COM

MÉTODO PUT

Agora vamos implementar a função para atualizar uma tarefa na base de dados. Para isso, não há segredos, basta utilizar a função `Task.update()`, cujo primeiro parâmetro você inclui um objeto com dados a serem atualizados e, no segundo, um objeto com dados de consulta das tarefas que serão atualizadas. Essa função retorna um simples array com um número de atualizações realizadas na base.

Mas esse dado não será de grande utilidade para nossa aplicação. Vamos forçar uma resposta de status `204 - No Content`, por meio da função `res.sendStatus(204)`, que significa que a requisição teve sucesso, porém não retornou conteúdo como resposta. Veja como fica essa implementação:

```
app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findOne({where: req.params})
      .then(result => {
        if (result) {
          res.json(result);
        } else {
          res.sendStatus(404);
        }
      })
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
  .put((req, res) => {
    Tasks.update(req.body, {where: req.params})
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
```

```
    });
  })
```

Assim como a função `Tasks.create()`, `Tasks.update()` faz uma limpeza dos campos que não existem no próprio modelo, então não há problemas em enviar o `req.body` diretamente.

6.7 EXCLUINDO UMA TAREFA COM MÉTODO DELETE

Para finalizar, temos de implementar a função de exclusão de tarefas, e mais uma vez não há segredos aqui! Basta utilizar a função `Tasks.destroy()` e passar em seu argumento um objeto com dados para consultar qual tarefa será excluída. Em nosso caso, vamos passar o `req.params.id` para implementar uma exclusão unitária das tarefas e, como resposta de sucesso, também será usado o status `204 - No Content`, via função `res.sendStatus(204)`. Veja como fica:

```
app.route("/tasks/:id")
  .all((req, res, next) => {
    delete req.body.id;
    next();
  })
  .get((req, res) => {
    Tasks.findOne({where: req.params})
      .then(result => {
        if (result) {
          res.json(result);
        } else {
          res.sendStatus(404);
        }
      })
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
```

```
  .put((req, res) => {
    Tasks.update(req.body, {where: req.params})
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  })
  .delete((req, res) => {
    Tasks.destroy({where: req.params})
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
```

E assim terminamos a implementação do CRUD de tarefas em nossa API.

6.8 REFACTORING NO MIDDLEWARE

Para evitar duplicidade de código, aplicaremos um simples *refactoring* migrando a lógica repetida da função `app.all()` para um middleware do Express, por meio do uso da função `app.use()` no arquivo `libs/middlewares.js`. Para aplicar esse refactoring, primeiro ainda no arquivo `routes/tasks.js`, remova as funções `app.all()`, deixando o código com a seguinte estrutura:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;

  app.route("/tasks")
    .get((req, res) => {
      // Lógica do GET /tasks
    })
    .post((req, res) => {
      // Lógica do POST /tasks
    });
  app.route("/tasks/:id")
```

```
    .get((req, res) => {
      // Lógica do GET /tasks/1
    })
    .put((req, res) => {
      // Lógica do PUT /tasks/1
    })
    .delete((req, res) => {
      // Lógica do DELETE /tasks/1
    });
};
```

Para que nossa API seja capaz de receber dados em formato JSON, será necessário incluir o middleware `body-parser`, que é responsável por fazer parser de objetos JSON, para isso execute:

```
npm install body-parser@1.15.0 --save
```

Após instalá-lo, abra e edite o `libs/middlewares.js`, e inclua no final dos middlewares o `app.use(bodyParser.json())` e principalmente a lógica de exclusão do `req.body.id`:

```
import bodyParser from "body-parser";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(bodyParser.json());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
};
```

6.9 IMPLEMENTANDO ROTAS PARA GESTÃO DE USUÁRIOS

Também temos de criar as rotas para gestão básica de usuários, afinal, sem eles, não será possível gerenciar tarefas, não é?

Nosso CRUD de usuários não terá muitas novidades. Somente uma, na função `Users.findById()` será incluído a propriedade `attributes: ["id", "name", "email"]`, que permitirá a execução da seguinte query SQL:

```
SELECT id, name, email FROM users;
```

Outro detalhe importante, as rotas de usuário não realizará um CRUD completo, pois terá apenas as lógicas para cadastrar, buscar e excluir um usuário. Cada rota será chamada diretamente por seu respectivo método do HTTP. Para codificá-lo, crie o arquivo `routes/users.js`, com o seguinte código:

```
module.exports = app => {
  const Users = app.db.models.Users;

  app.get("/users/:id", (req, res) => {
    Users.findById(req.params.id, {
      attributes: ["id", "name", "email"]
    })
    .then(result => res.json(result))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  });
  app.delete("/users/:id", (req, res) => {
    Users.destroy({where: {id: req.params.id} })
      .then(result => res.sendStatus(204))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
  app.post("/users", (req, res) => {
    Users.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
};
```

Por enquanto não será usado a função `app.route()` , mas no próximo capítulo *Autenticando usuários na API*, essa função será usada em alguns pontos dessa rota, para consultar e excluir somente quando o usuário estiver logado no sistema.

6.10 TESTANDO ROTAS COM POSTMAN

Para testar essas modificações, reinicie a aplicação, abra o browser e utilize algum aplicativo cliente REST, pois será necessário para testar os métodos `POST` , `PUT` e `DELETE` . Para simplificar, recomendo a utilização do Postman, que é uma extensão para Google Chrome muito completo e fácil de usar.

Para instalá-lo acesse: https://www.getpostman.com. Então, clique no botão "*Get it now - it's free!*". Após sua instalação, na tela de Apps do Chrome, acesse o ícone do aplicativo Postman.

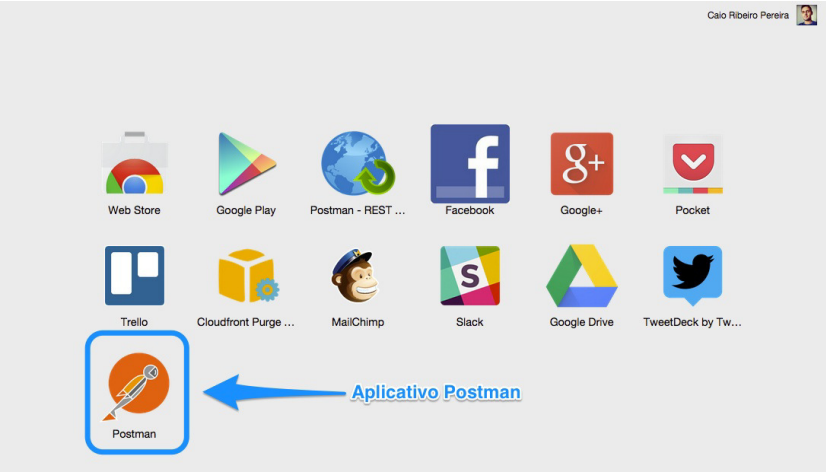


Figura 6.1: Postman Rest Client

Surgirá uma tela para fazer login, porém você não precisa fazer login no sistema. Para pular e ir direto para a tela principal, clique no botão "*Go to the app*".

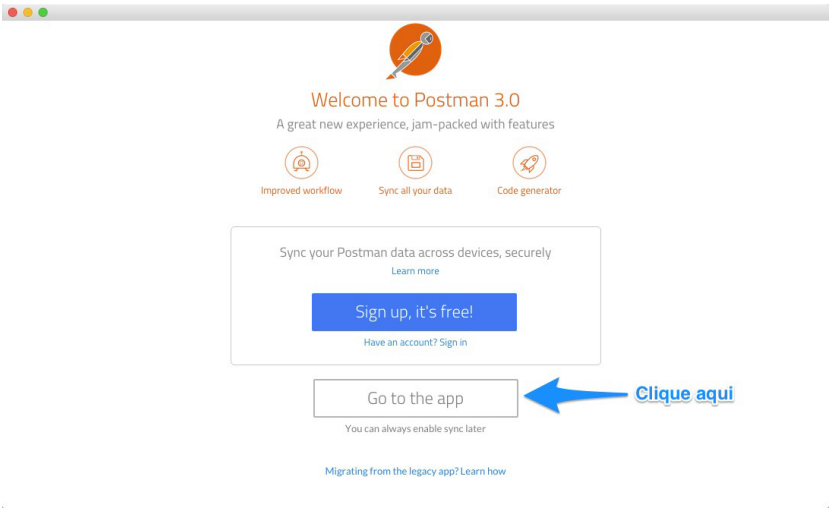

Welcome to Postman 3.0
A great new experience, jam-packed with features
Improved workflow
Sync all your data
Code generator
Sync your Postman data across devices, securely
Learn more
Sign up, it's free!
Have an account? Sign in
Go to the app
Clique aqui
You can always enable sync later
Migrating from the legacy app? Learn how

Figura 6.2: Abrindo o Postman

Para testar os endpoints, realize os seguintes testes:

1. Escolha o método `POST` com endereço http://localhost:3000/tasks;
2. Clique no menu `Body` escolha a opção `raw` , e altere o formato `Text` para `JSON (application/json)` ;
3. Crie o JSON `{"title": "Durmir"}` e clique no botão `Send` ;
4. Modifique o mesmo JSON para `{"title": "Estudar"}` e clique novamente no `Send` ;

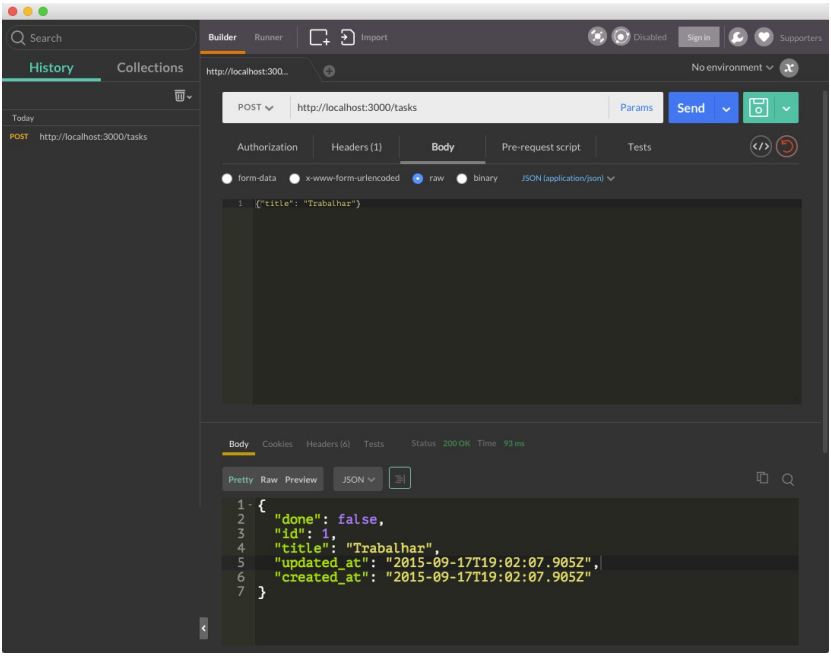


Figura 6.3: Cadastrando tarefa 'Trabalhar'

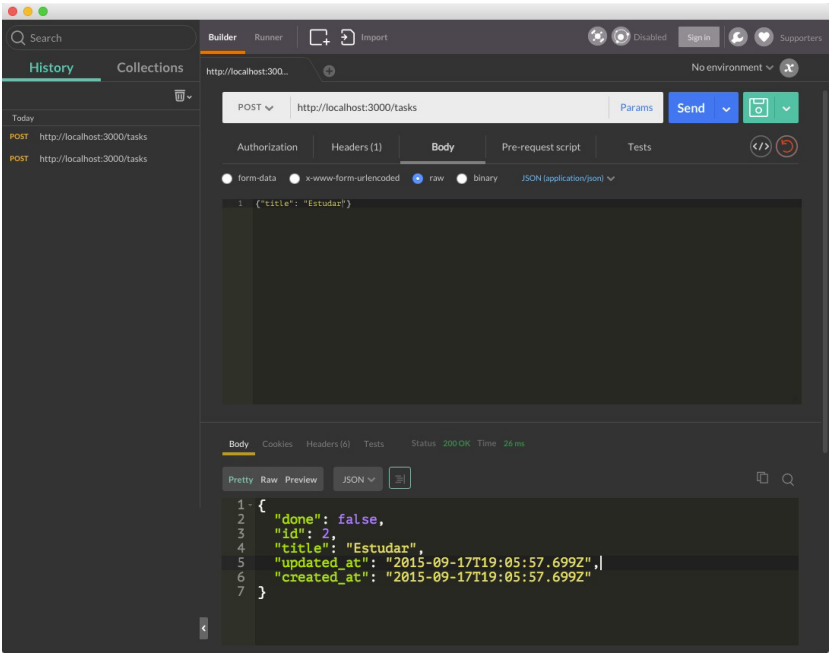


Figura 6.4: Cadastrando tarefa 'Estudar'

Com esse procedimento, você cadastrou duas tarefas, e agora é possível explorar as outras rotas de gestão de tarefas. Para testar os demais endpoints, você pode seguir essa simples lista:

- Método `GET`, rota http://localhost:3000/tasks.
- Método `GET`, rota http://localhost:3000/tasks/1.
- Método `GET`, rota http://localhost:3000/tasks/2.
- Método `PUT`, rota http://localhost:3000/tasks/1, body `{"title": "Trabalhar"}`.
- Método `DELETE`, rota http://localhost:3000/tasks/2.

Você também pode testar as rotas de usuários, se quiser. Você pode seguir essa simples bateria de testes:

- Método `POST`, rota http://localhost:3000/users, body `{"name": "John", "email": "john@connor.net", "password": "123"}`.
- Método `GET`, rota http://localhost:3000/users/1.
- Método `DELETE`, rota http://localhost:3000/users/1.

Conclusão

Parabéns para você que chegou até aqui vivo! Já temos um esboço significativo de uma API RESTful, ou seja, já é possível construir aplicações cliente para consumir os recursos, tarefas e usuários.

Continue lendo, pois no próximo capítulo vamos implementar funcionalidades importantes sobre autenticação de usuários em nossa API. *See ya!*

Capítulo 7

AUTENTICANDO USUÁRIOS NA API

Nossa API já tem um CRUD de tarefas que, graças ao framework Sequelize, está integrado a um banco de dados SQL – no nosso caso, o SQLite3. Já implementamos também suas rotas por meio das principais funções de roteamento e middlewares do framework Express.

Neste capítulo, vamos explorar os principais conceitos e implementações de autenticação de usuários na API. Afinal, esta é uma etapa importante e necessária para garantir que os usuários gerenciem suas tarefas com segurança na aplicação.

7.1 INTRODUÇÃO AO PASSPORT E JWT

Sobre o Passport

Existe um módulo para Node.js muito bacana e fácil de trabalhar com autenticações de usuário, seu nome é Passport.

O Passport é um framework extremamente flexível e modular. Ele permite trabalhar com as principais estratégias de autenticação: **Basic & Digest**, **OpenID**, **OAuth**, **OAuth 2.0** e **JWT**. E também

permite trabalhar com autenticação via serviços externos, como por exemplo, autenticação por Facebook, Google+, Twitter e muito mais. Aliás, em seu site oficial, **existe uma lista com +300 estratégias de autenticações**, que foram criados e adaptados por vários desenvolvedores.

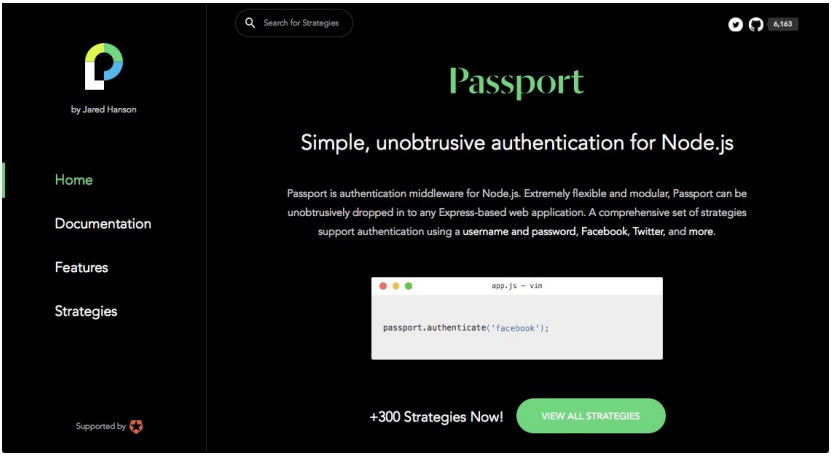

Search for Strategies
6,163
by Jared Hanson
Passport
Simple, unobtrusive authentication for Node.js
Home
Documentation
Features
Strategies
Passport is authentication middleware for Node.js. Extremely flexible and modular, Passport can be unobtrusively dropped in to any Express-based web application. A comprehensive set of strategies support authentication using a username and password, Facebook, Twitter, and more.
app.js — vim
passport.authenticate('facebook');
Supported by
+300 Strategies Now!
VIEW ALL STRATEGIES

Figura 7.1: Homepage do Passport

Seu site oficial é http://passportjs.org.

Sobre o JWT

O JWT (*JSON Web Tokens*) é uma estratégia bem simples e segura para autenticação de APIs RESTful. Ela é um *open standard* para autenticações de aplicações web baseado no tráfego de tokens em formato JSON, entre o cliente e servidor. Seu fluxo de autenticação funciona da seguinte maneira:

1. Cliente realiza uma requisição uma única vez, enviando suas credenciais de login e senha;

2. Servidor valida as credenciais e, se tudo estiver certo, ele retorna para o cliente um JSON com token que encodifica os dados de um usuário logado no sistema;
3. Cliente, ao receber esse token, pode armazená-lo da maneira que quiser, seja via LocalStorage, Cookie ou outros mecanismos de armazenamento client-side;
    - Toda vez que o cliente acessar uma rota que necessita autenticação, ele terá de apenas enviar esse token para a API autenticar e liberar o consumo de dados;
4. Servidor sempre validará esse token para permitir ou não uma requisição de cliente.

Para detalhes mais específicos sobre o JWT, acesse http://jwt.io.

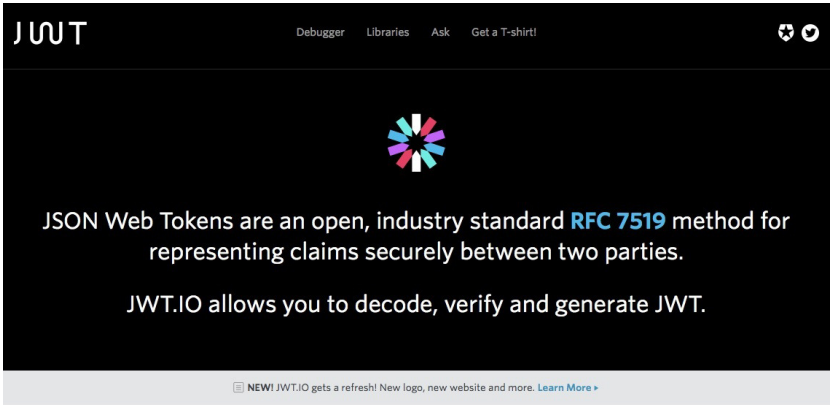

JWT
Debugger
Libraries
Ask
Get a T-shirt!
JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.
JWT.IO allows you to decode, verify and generate JWT.
NEW! JWT.IO gets a refresh! New logo, new website and more. Learn More

Figura 7.2: Homepage do JWT

Atenção

Vamos usar o Passport e JWT em nossa API, afinal, o Passport é um módulo muito flexível e poderoso quando o assunto é autenticações de usuários. Ele possui uma lista extensa com suporte a **+300** diversas estratégias de autenticação, e o JWT é uma delas também.

A adoção do JWT é muito viável para o nosso projeto, pois é considerado um mecanismo leve e prático no tráfego de dados. Afinal, vamos trabalhar com *tokens* em formato JSON, e esse formato já está sendo largamente usado em nosso projeto. O JWT é também um mecanismo fácil, seguro e sua adoção é muito recomendada para aplicações do tipo API RESTful, que é o nosso caso.

7.2 INSTALANDO PASSPORT E JWT NA API

Nessa brincadeira, vamos usar os seguintes módulos:

- **Passport:** como mecanismo de autenticação;
- **Passport JWT:** extensão do JWT para uso como estratégia de autenticação no Passport;
- **JWT Simple:** biblioteca para encodificação/decodificação de tokens do JWT.

Agora instale-os rodando o comando:

```
npm install passport@0.3.2 passport-jwt@2.0.0
                                        jwt-simple@0.4.1 --save
```

Para começar essa implementação, primeiro vamos adicionar 3 novos itens de configuração do JWT. Edite o arquivo `libs/config.js` e inclua, no final, os seguintes atributos:

```
module.exports = {
  database: "ntask",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    define: {
      underscored: true
    }
  },
  jwtSecret: "Nta$K-AP1",
  jwtSession: {session: false}
};
```

O campo `jwtSecret` mantém uma string de chave secreta que servirá como base para *encode/decode* de tokens. É recomendável que essa string seja complexa, utilizando diversos caracteres diferentes.

Jamais compartilhe ou divulgue essa chave secreta em público, pois, se ela vazar, você deixará sua aplicação vulnerável a invasão, permitindo que uma pessoa má intencionada acesse o sistema e gere tokens autenticáveis, sem informar as credenciais de login e senha no sistema.

Para finalizar, o último campo incluído é o `jwtSession`, que possui o objeto `{session: false}`. Esse item será utilizado para informar ao Passport que a autenticação não terá sessão de usuário.

7.3 IMPLEMENTANDO AUTENTICAÇÃO JWT

Agora que temos as configurações do Passport e JWT prontas, vamos implementar as regras de como um cliente será autenticado na aplicação. Para começar, implementaremos as regras de autenticação, que também terá funções de middlewares do Passport para usarmos nas rotas da API. Esse código terá um middleware e duas funções. O middleware será executado no momento que ele for carregado na aplicação, e ele basicamente recebe em seu callback um `payload`, que é um JSON decodificado pela chave secreta `cfg.jwtSecret`. Esse `payload` terá o atributo `id`, que será um `id` de usuário a ser consultado pela função `Users.findById(payload.id)`. Como esse middleware será frequentemente acessado, para evitar *overhead* na aplicação, vamos enviar um objeto simples contendo apenas o `id` e `email` do usuário autenticado, por meio da função callback:

```
done(null, {id: user.id, email: user.email});
```

A lógica desse middleware é injetada via função `passport.use(strategy)`. Para finalizar, vamos retornar 2 funções do Passport para serem utilizadas no decorrer da aplicação. Elas são as funções `initialize` (inicializa o Passport) e `authenticate` (usada para autenticar acesso a uma rota).

Para entender melhor essa implementação, crie na pasta raiz o arquivo `auth.js`, com esse código:

```
import passport from "passport";
import {Strategy, ExtractJwt} from "passport-jwt";

module.exports = app => {
  const Users = app.db.models.Users;
  const cfg = app.libs.config;
  const params = {
    secretOrKey: cfg.jwtSecret,
    jwtFromRequest: ExtractJwt.fromAuthHeader()
```

```
  };
  const strategy = new Strategy(params, (payload, done) => {
    Users.findById(payload.id)
      .then(user => {
        if (user) {
          return done(null, {
            id: user.id,
            email: user.email
          });
        }
        return done(null, false);
      })
      .catch(error => done(error, null));
  });
  passport.use(strategy);
  return {
    initialize: () => {
      return passport.initialize();
    },
    authenticate: () => {
      return passport.authenticate("jwt", cfg.jwtSession);
    }
  };
};
```

Para carregar o `auth.js` no início da aplicação, edite o código `index.js` da seguinte maneira:

```
import express from "express";
import consign from "consign";

const app = express();

consign()
  .include("libs/config.js")
  .then("db.js")
  .then("auth.js")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);
```

Para inicializar o Passport no Express, edite o

`libs/middlewares.js` e inclua o middleware `app.use(app.auth.initialize())`. Veja a seguir onde incluí-lo:

```
import bodyParser from "body-parser";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
};
```

7.4 GERANDO TOKENS PARA USUÁRIOS AUTENTICADOS

Para finalizar a implementação de autenticação JWT em nossa aplicação, vamos agora preparar o modelo `Users` para criptografia de senha de usuário. Também criaremos uma rota para gerar tokens para os usuários que se autenticarem com seu login e senha no sistema, e faremos um *refactoring* nas rotas de tarefas e usuários para que suas consultas usem corretamente o `id` de usuário autenticado. Com isso, finalizaremos essa etapa de autenticação, deixando nossa aplicação mais segura e confiável.

A criptografia de senha dos usuários será realizada pelo módulo `bcrypt`. Para isso, instale-o rodando o comando:

```
npm install bcrypt@0.8.5 --save
```

Agora, vamos editar o modelo `Users`. Nele incluiremos uma função de `hooks`, que são funções executáveis antes ou depois de

uma operação no banco de dados. No nosso caso, vamos incluir uma função para ser executada antes de cadastrar um novo usuário, por meio do uso da função `beforeCreate`. Vamos utilizar o `bcrypt` para criptografar a senha do usuário antes de salvá-la na tabela de usuários.

Também será incluída uma nova função dentro de `classMethods`. Ela será usada para comparar se uma senha informada é igual a uma senha criptografada do usuário. Para codificar essas regras, edite o `models/users.js` com a seguinte lógica:

```
import bcrypt from "bcrypt";

module.exports = (sequelize, DataType) => {
  const Users = sequelize.define("Users", {
    // Os campos desse modelo foram criados no capítulo 5
  }, {
    hooks: {
      beforeCreate: user => {
        const salt = bcrypt.genSaltSync();
        user.password = bcrypt.hashSync(user.password, salt);
      }
    },
    classMethods: {
      associate: models => {
        Users.hasMany(models.Tasks);
      },
      isPassword: (encodedPassword, password) => {
        return bcrypt.compareSync(password, encodedPassword);
      }
    }
  });
  return Users;
};
```

Com essas modificações implementadas no modelo `Users`, podemos agora codificar o novo endpoint `/token`. Ele será responsável por gerar um token encodificado com um `payload`,

dado o usuário que enviar o e-mail e senha correto por meio do corpo da requisição (`req.body.email` e `req.body.password`).

O `payload` terá apenas o `id` de usuário. A geração do token ocorre pelo módulo `jwt-simple` utilizando sua função `jwt.encode(payload, cfg.jwtSecret)` que, obrigatoriamente, usará a mesma chave secreta `jwtSecret` , que foi criada no arquivo `libs/config.js` . Qualquer erro gerado nessa rota será tratado através da resposta de status `401 - Unauthorized` do HTTP, com a função `res.sendStatus(401)` .

Para incluir essa regra de geração de tokens, crie o arquivo `routes/token.js` com o seguinte código:

```
import jwt from "jwt-simple";

module.exports = app => {
  const cfg = app.libs.config;
  const Users = app.db.models.Users;
  app.post("/token", (req, res) => {
    if (req.body.email && req.body.password) {
      const email = req.body.email;
      const password = req.body.password;
      Users.findOne({where: {email: email}})
        .then(user => {
          if (Users.isPassword(user.password, password)) {
            const payload = {id: user.id};
            res.json({
              token: jwt.encode(payload, cfg.jwtSecret)
            });
          } else {
            res.sendStatus(401);
          }
        })
        .catch(error => res.sendStatus(401));
    } else {
      res.sendStatus(401);
    }
  });
};
```

Já temos a lógica de autenticação de usuários e também a de validação do token. Para finalizar, usaremos a função `app.auth.authenticate()`, que valida os tokens enviados pelos clientes e libera (ou não) o acesso a uma determinada rota da aplicação. Para isso, edite o arquivo `routes/tasks.js` e inclua a função middleware `all(app.auth.authenticate())` no início das duas rotas. Veja a seguir como fica:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;

  app.route("/tasks")
    .all(app.auth.authenticate())
    .get((req, res) => {
      // Lógica do GET /tasks
    })
    .post((req, res) => {
      // Lógica do POST /tasks
    });

  app.route("/tasks/:id")
    .all(app.auth.authenticate())
    .get((req, res) => {
      // Lógica do GET /tasks/1
    })
    .put((req, res) => {
      // Lógica do PUT /tasks/1
    })
    .delete((req, res) => {
      // Lógica do DELETE /tasks/1
    });
};
```

Quando um cliente envia um token válido, o seu acesso a uma rota é autenticado com sucesso e, consequentemente, surge o objeto `req.user` para usá-lo na lógica das rotas. Esse objeto é criado somente quando a lógica do `auth.js` retorna um usuário autenticado, ou seja, somente quando a função a seguir retorna um usuário válido:

```
Users.findById(payload.id)
  .then(user => {
    if (user) {
      // Lembra dessa função?
      return done(null, {
        id: user.id,
        email: user.email
      });
    }
    return done(null, false);
  })
  .catch(error => done(error, null));
```

A função `done()` envia os dados de usuário autenticado e as rotas autenticada recebem esses dados através do objeto `req.user`. No nosso caso, esse objeto terá apenas os atributos: `id` e `email`.

Para garantir um acesso correto nos dados do modelo `Tasks`, vamos fazer alguns *refactorings* em todas as funções de acesso à base de dados existentes nas rotas `/tasks` e `/tasks/:id`. Para isso, edite o `routes/tasks.js` e, dentro das rotas de `app.route("/tasks")`, faça a seguinte modificação:

```
app.route("/tasks")
  .all(app.auth.authenticate())
  .get((req, res) => {
    Tasks.findAll({
      where: { user_id: req.user.id }
    })
    .then(result => res.json(result))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  })
  .post((req, res) => {
    req.body.user_id = req.user.id;
    Tasks.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
```

```
    });
});
```

Ainda no mesmo arquivo, faça as modificações nas queries das rotas internas da função `app.route("/tasks/:id")`:

```
app.route("/tasks/:id")
  .all(app.auth.authenticate())
  .get((req, res) => {
    Tasks.findOne({ where: {
      id: req.params.id,
      user_id: req.user.id
    }})
    .then(result => {
      if (result) {
        return res.json(result);
      }
      return res.sendStatus(404);
    })
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  })
  .put((req, res) => {
    Tasks.update(req.body, { where: {
      id: req.params.id,
      user_id: req.user.id
    }})
    .then(result => res.sendStatus(204))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  })
  .delete((req, res) => {
    Tasks.destroy({ where: {
      id: req.params.id,
      user_id: req.user.id
    }})
    .then(result => res.sendStatus(204))
    .catch(error => {
      res.status(412).json({msg: error.message});
    });
  });
```

Para finalizar esse *refactoring* de acesso aos recursos por meio de usuários autenticados, vamos adaptar alguns trechos de código das rotas de usuários. Basicamente mudaremos a maneira como se faz uma consulta e exclusão de usuário, para que somente seja realizada via `id` do usuário autenticado.

Então, neste caso, não será mais necessário passar um `id` através do parâmetro da rota, já que agora a rota `/users/:id` será apenas `/user` (no singular mesmo, pois estaremos lidando com um único usuário logado). Somente a consulta e exclusão terão um middleware de autenticação, logo, ambos poderão se agrupar via função `app.route("/user")` para usarem o middleware da função `all(app.auth.authenticate())`. No lugar do `req.params.id`, vamos usar `req.user.id`, para garantir que seja usado o `id` de um usuário autenticado.

Para entender melhor como será essa lógica, edite o arquivo `routes/users.js` e faça as modificações a seguir:

```
module.exports = app => {
  const Users = app.db.models.Users;

  app.route("/user")
    .all(app.auth.authenticate())
    .get((req, res) => {
      Users.findById(req.user.id, {
        attributes: ["id", "name", "email"]
      })
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
    })
    .delete((req, res) => {
      Users.destroy({where: {id: req.user.id} })
        .then(result => res.sendStatus(204))
        .catch(error => {
```

```
          res.status(412).json({msg: error.message});
        });
    });
  // Essa rota não precisa de autenticação
  app.post("/users", (req, res) => {
    Users.create(req.body)
      .then(result => res.json(result))
      .catch(error => {
        res.status(412).json({msg: error.message});
      });
  });
};
```

Conclusão

Parabéns! Finalizamos uma etapa extremamente importante da aplicação. Dessa vez, os dados das tarefas serão consultados corretamente por um usuário autenticado na aplicação. Graças ao JWT, foi possível implementar um mecanismo de autenticação segura, que evita o tráfego frequente de senhas entre cliente e servidor.

Até agora, só foi implementado todo o back-end da aplicação, e ainda não criamos uma aplicação final que use todo poder de nossa API. Mas fique tranquilo, há muitas surpresas boas nos próximos capítulos que vão deixá-lo muito feliz, apenas continue lendo!

Capítulo 8

TESTANDO A APLICAÇÃO — PARTE 1

8.1 INTRODUÇÃO AO MOCHA

Criar testes automatizados é algo largamente adotado no desenvolvimento de sistemas. Existem diversos tipos de testes: unitário, funcional, de aceitação, entre outros. Neste capítulo, focaremos apenas no teste de aceitação, que no nosso caso visa testar as respostas de sucesso e erros das rotas de nossa API.

Para criar e executar os testes, vamos usar o TestRunner chamado Mocha, que é um módulo muito popular para o Node.js.

Figura 8.1: Mocha – TestRunner para Node.js

O Mocha foi possui as seguintes características:

- Testes no estilo TDD;
- Testes no estilo BDD;
- Cobertura de código com relatório para HTML;
- Resultado dos testes customizado;
- Teste para funções assíncronas;
- Facilmente integrado com os módulos `should`, `assert` e `chai`.

Praticamente, ele é um ambiente completo para desenvolvimento de testes para Node.js. Seu site oficial é https://mochajs.org.

8.2 CONFIGURANDO AMBIENTE PARA TESTES

Para configurarmos nosso ambiente de testes, primeiro configuraremos uma nova base de dados que será usada apenas para brincarmos com dados *fakes* pelos testes. Essa prática é largamente utilizada para garantir que uma aplicação seja facilmente trabalhada em múltiplos ambientes. Por enquanto, nossa API possui apenas configurações de um único ambiente, pois até agora, todos os exemplos foram desenvolvidos no ambiente de desenvolvimento.

Para habilitarmos o suporte a múltiplos ambientes, vamos renomear o atual arquivo `libs/config.js` para `libs/config.development.js` e, em seguida, vamos criar o arquivo `libs/config.test.js`. O único parâmetro novo nesse arquivo é o `logging: false`, que desabilita os logs de comandos SQL no terminal. Será necessário desabilitarmos esses logs para não gerar um report de testes confuso. A seguir, veja como fica

esse arquivo:

```
module.exports = {
  database: "ntask_test",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    logging: false,
    define: {
      underscored: true
    }
  },
  jwtSecret: "NTASK_TEST",
  jwtSession: {session: false}
};
```

Agora, temos dois arquivos de configurações, cada qual contém dados específicos para seu respectivo ambiente. Para que a nossa aplicação carregue as configurações de acordo com o ambiente, vamos realizar alguns *refactorings* para que ela identifique em qual ambiente ela se encontra. Neste caso, vamos usar o `process.env`, que basicamente retorna um objeto com diversas variáveis de ambiente do sistema operacional.

Uma boa prática em projetos Node.js é trabalhar com a variável `process.env.NODE_ENV` e, com base no seu valor retornado, a nossa aplicação terá de carregar configurações para o ambiente `test` ou `development` (por default, será sempre `development`, caso o retorno dessa variável seja nula ou uma string vazia).

Com base nisso, recriaremos o arquivo `libs/config.js` para que ele carregue a configuração de acordo com o valor da variável de ambiente do sistema operacional. Veja como deve ficar:

```
module.exports = app => {
  const env = process.env.NODE_ENV;
  if (env) {
    return require(`./config.${env}.js`);
  }
  return require("./config.development.js");
};
```

Em nosso projeto, vamos explorar apenas a criação de testes de aceitação, que serão testes em cima do comportamento e resultado dos endpoints da API. Para a criação deles, usaremos os seguintes módulos:

- `babel-register` : para rodar código ES6;
- `mocha` : para rodar os testes;
- `chai` : para utilizar uma interface BDD nos testes;
- `supertest` : para realizar requisições na API;

Todos esses módulos serão instalados como um `devDependencies` no `package.json` , para usá-lo apenas como dependência de desenvolvimento e testes. Isso você faz usando a flag `--save-dev` . Veja o comando a seguir:

```
npm install babel-register@6.5.2 mocha@2.4.5 chai@3.5.0
                                    supertest@1.2.0 --save-dev
```

Agora, vamos encapsular a execução do `mocha` por meio do comando `npm test` , para que ele execute internamente o comando `NODE_ENV=test mocha test/**/*.js` . Para implementar esse novo comando, edite o `package.json` :

```
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "API de gestão de tarefas",
  "main": "index.js",
  "scripts": {
    "start": "babel-node index.js",
```

```
    "test": "NODE_ENV=test mocha test/**/*.js"
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0",
    "bcrypt": "^0.8.5",
    "body-parser": "^1.15.0",
    "consign": "^0.1.2",
    "express": "^4.13.4",
    "jwt-simple": "^0.4.1",
    "passport": "^0.3.2",
    "passport-jwt": "^2.0.0",
    "sequelize": "^3.19.2",
    "sqlite3": "^3.1.1"
  },
  "devDependencies": {
    "babel-register": "^6.5.2",
    "chai": "^3.5.0",
    "mocha": "^2.4.5",
    "supertest": "^1.2.0"
  }
}
```

Se você estiver usando Windows, mude o comando `NODE_ENV=test mocha test/**/*.js` para `SET NODE_ENV=test && mocha test/**/*.js` dentro de `scripts.test`, veja como fica:

```
"scripts": {
  "start": "babel-node index.js",
  "test": "SET NODE_ENV=test && mocha test/**/*.js"
},
```

Em seguida, exportaremos nossa API para que ela seja iniciada ao executar os testes com o Mocha. Para fazer isso, basta incluir no final do `index.js` a função `module.exports = app`. Também vamos desabilitar alguns logs gerados pelo módulo `consign` pelo trecho `consign({verbose: false})` para não poluir o report dos testes.

```
import express from "express";
import consign from "consign";

const app = express();

consign({verbose: false})
  .include("libs/config.js")
  .then("db.js")
  .then("auth.js")
  .then("libs/middlewares.js")
  .then("routes")
  .then("libs/boot.js")
  .into(app);

module.exports = app;
```

Agora, a aplicação será iniciada internamente pelo módulo `supertest`. Para evitar que o servidor inicie duas vezes em ambiente de testes, vamos modificar o `libs/boot.js` para que não seja iniciada quando `process.env.NODE_ENV` estiver com valor `"test"`.

Para alterar isso, edite o `libs/boot.js` com esse código:

```
module.exports = app => {
  if (process.env.NODE_ENV !== "test") {
    app.db.sequelize.sync().done(() => {
      app.listen(app.get("port"), () => {
        console.log(`NTask API - porta ${app.get("port")}`);
      });
    });
  }
}
```

Para terminar o nosso setup de ambiente de testes, prepararemos algumas configurações específicas do Mocha, para que ele carregue o servidor da API e os módulos `chai` e `supertest`, como variáveis globais. O motivo disso é agilizar a execução dos testes, afinal, cada um carregaria esses módulos e, se

centralizarmos tudo isso em um único arquivo, economizaríamos alguns milissegundos de execução dos testes. Para implementar essa boa prática, crie o arquivo `test/helpers.js` :

```
import supertest from "supertest";
import chai from "chai";
import app from "../index.js";

global.app = app;
global.request = supertest(app);
global.expect = chai.expect;
```

Em seguida, vamos criar um simples arquivo que permite incluir parâmetros de configurações para o comando `mocha` . Este será responsável por carregar o `test/helpers.js` , e terá também uma flag `--reporter spec` para usar report mais detalhado dos testes que são executados. Depois, vamos incluir a flag `--compilers js:babel-register` para que o Mocha utilize o módulo `babel` para reconhecer e executar os códigos dos testes no padrão JavaScript ES6.

Por último, será incluída a flag `--slow 5000` para que a bateria de testes demorem 5 segundos para iniciar (tempo suficiente para o servidor de API carregar as tabelas do Sequelize corretamente). Crie o arquivo `test/mocha.opts` com oa seguintes parâmetros:

```
--require test/helpers
--reporter spec
--compilers js:babel-register
--slow 5000
```

Criando o primeiro teste

Pronto! Terminamos o setup básico para execução de testes com Mocha. Vamos testar alguma coisa? Que tal testarmos o

`routes/index.js` ? Ele é muito simples de se testar: basicamente vamos testar o JSON de retorno dele, comparando se o resultado é igual ao JSON `{status: "NTask API"}` .

Para criar nosso primeiro teste, realizaremos uma requisição `GET /` . Por meio da função `request.get("/")` , será validado se a requisição retorna `status 200` . Para finalizar, é feita uma comparação entre objeto `req.body` com o objeto `expected` para validar se ambos são iguais, via função `expect(res.body).to.eql(expected)` .

Para implementar esse teste, crie o arquivo `test/routes/index.js` com os seguintes códigos:

```
describe("Routes: Index", () => {
  describe("GET /", () => {
    it("returns the API status", done => {
      request.get("/")
        .expect(200)
        .end((err, res) => {
          const expected = {status: "NTask API"};
          expect(res.body).to.eql(expected);
          done(err);
        });
    });
  });
});
```

Para executar nosso primeiro teste, basta rodar o comando:

```
npm test
```

Após a sua execução, você terá um resultado semelhante a esta figura:

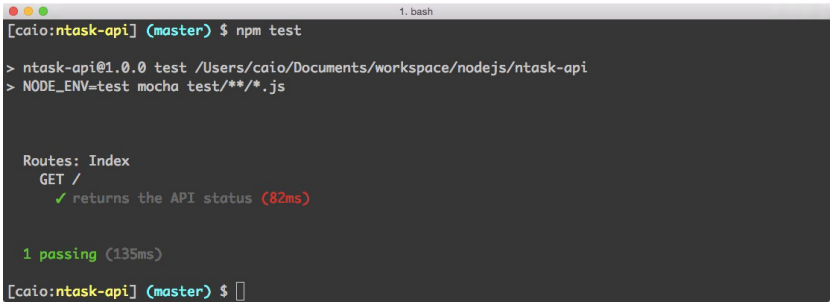


Figura 8.2: Executando o primeiro teste

8.3 TESTANDO ENDPOINT DE AUTENTICAÇÃO DA API

Sem enrolações! Nesta seção, vamos implementar testes e mais testes sobre os endpoints da nossa aplicação. Para começar, testaremos o endpoint `routes/token.js` , que é responsável por gerar tokens para os usuários autenticados.

Basicamente, esse endpoint terá 4 testes que vão validar:

- Requisição autenticada por um usuário válido;
- Requisição com e-mail válido informando senha incorreta;
- Requisição informando um e-mail não cadastrado;
- Requisição sem e-mail e sem senha.

Crie o teste `test/routes/token.js` com a seguinte estrutura:

```
describe("Routes: Token", () => {
  const Users = app.db.models.Users;
  describe("POST /token", () => {
    beforeEach(done => {
      // Código de pré-teste
    });
    describe("status 200", () => {
```

```
      it("returns authenticated user token", done => {
        // Código do teste...
      });
    });
    describe("status 401", () => {
      it("throws error when password is incorrect", done => {
        // Código do teste...
      });
      it("throws error when email not exist", done => {
        // Código do teste...
      });
      it("throws error when email and password are blank",
                                                          done => {
        // Código do teste...
      });
    });
  });
});
```

Para iniciar, vamos codar a lógica interna da função `beforeEach()`. Essa função é executada antes de cada teste, e basicamente terá de cadastrar um usuário na base. Para isso, vamos usar o modelo `app.db.models.Users` e suas funções: `Users.destroy({where: {}})` para limpar a tabela de usuários, e `Users.create` para cadastrar um novo em seguida, a cada execução dos testes. Isso vai permitir testar utilizando um usuário válido.

```
beforeEach(done => {
  Users
    .destroy({where: {}})
    .then(() => Users.create({
      name: "John",
      email: "john@mail.net",
      password: "12345"
    }))
    .then(() => done());
});
```

Agora, vamos implementar teste a teste. Começaremos com o

primeiro teste, que retorna um caso de sucesso. Vamos usar a função `request.post("/token")` para fazer uma requisição do token, já enviando o e-mail e a senha de um usuário válido através da função `send()`. A função `expect(200)` indica que a resposta esperada é por meio do `status 200` do HTTP.

Para finalizar o teste, no callback da função `end(err, res)`, é validado se o objeto `res.body` retorna o atributo `token` via função `expect(res.body).to.include.keys("token")`. Para encerrar um teste, é obrigatória a execução do callback `done()` no final do teste, pois é ela a função que o finaliza.

Preferencialmente, sempre envie a variável `err` como parâmetro para essa função (`done(err)`), pois, caso ocorra um erro na requisição, serão exibidos no terminal os detalhes do erro ocorrido. Veja a seguir o código completo desse teste:

```
it("returns authenticated user token", done => {
  request.post("/token")
    .send({
      email: "john@mail.net",
      password: "12345"
    })
    .expect(200)
    .end((err, res) => {
      expect(res.body).to.include.keys("token");
      done(err);
    });
});
```

Em seguida, vamos testar o caso do envio de senha incorreta, esperando que ela retorne status 401 de acesso não autorizado. Esse teste será mais simples, pois basicamente vamos testar apenas se a requisição retornará erro de `status 401`, através da função `expect(401)`.

```
it("throws error when password is incorrect", done => {
  request.post("/token")
    .send({
      email: "john@mail.net",
      password: "SENHA_ERRADA"
    })
    .expect(401)
    .end((err, res) => {
      done(err);
    });
});
```

Também vamos implementar o teste de e-mail inexistente na tabela de usuários. As funções usadas nele são semelhantes ao teste anterior.

```
it("throws error when email not exist", done => {
  request.post("/token")
    .send({
      email: "EMAIL_ERRADO",
      password: "SENHA_ERRADA"
    })
    .expect(401)
    .end((err, res) => {
      done(err);
    });
});
```

E, para finalizar, vamos criar os testes de status 401, quando não é enviado um e-mail e nem uma senha. Este é mais simples ainda, pois não serão enviados parâmetros no corpo da requisição, basicamente ele é validado através da função `expect(401)` e ponto final.

```
it("throws error when email and password are blank", done => {
  request.post("/token")
    .expect(401)
    .end((err, res) => {
      done(err);
    });
});
```

Conclusão

Parabéns! Se você implementou até aqui e rodou novamente o comando `npm test`, você provavelmente terá um resultado semelhante a esta figura:

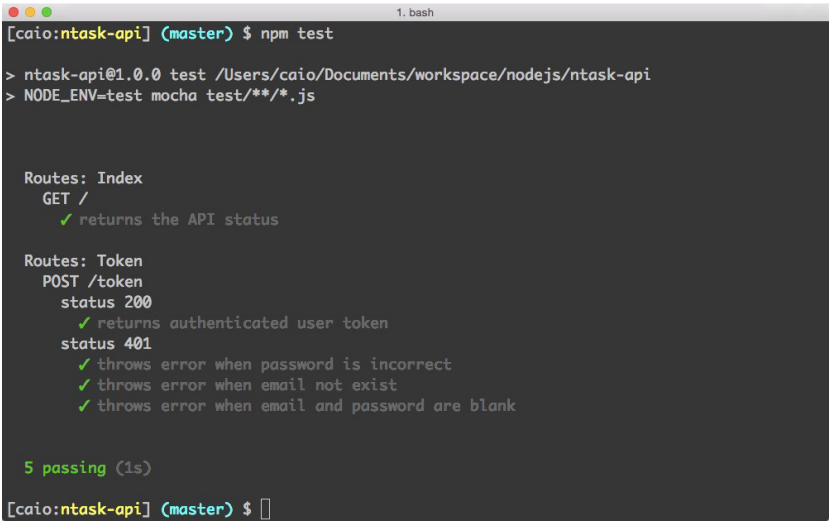


Figura 8.3: Resultado dos testes de autenticação

Continue lendo, pois este assunto de testes é um pouco extenso, e vamos continuá-lo no próximo capítulo, com a parte final da implementação dos testes nos endpoints da API.

Capítulo 9

TESTANDO A APLICAÇÃO — PARTE 2

Dando continuidade à implementação dos testes para nossa API, vamos agora focar nos testes para os recursos: tarefas e usuários.

9.1 TESTANDO OS ENDPOINTS DAS TAREFAS

Para testarmos os endpoints do recurso tarefas, teremos de fazer um pequeno contorno para burlar a autenticação JWT na aplicação. Afinal, será necessário para testarmos corretamente os resultados desse recurso e também dos demais que envolvam uma autenticação de usuário. Para começar, vamos criar a estrutura dos testes para tarefas.

Crie o arquivo `test/routes/tasks.js` com o seguinte layout:

```
import jwt from "jwt-simple";

describe("Routes: Tasks", () => {
  const Users = app.db.models.Users;
  const Tasks = app.db.models.Tasks;
  const jwtSecret = app.libs.config.jwtSecret;
  let token;
  let fakeTask;
```

```
  beforeEach(done => {
    // Código de testes
  });
  describe("GET /tasks", () => {
    describe("status 200", () => {
      it("returns a list of tasks", done => {
        // Código de testes
      });
    });
  });
  describe("POST /tasks/", () => {
    describe("status 200", () => {
      it("creates a new task", done => {
        // Código de testes
      });
    });
  });
  describe("GET /tasks/:id", () => {
    describe("status 200", () => {
      it("returns one task", done => {
        // Código de testes
      });
    });
    describe("status 404", () => {
      it("throws error when task not exist", done => {
        // Código de testes
      });
    });
  });
  describe("PUT /tasks/:id", () => {
    describe("status 204", () => {
      it("updates a task", done => {
        // Código de testes
      });
    });
  });
  describe("DELETE /tasks/:id", () => {
    describe("status 204", () => {
      it("removes a task", done => {
        // Código de testes
      });
    });
  });
});
```

Entrando em detalhes sobre como vamos burlar a autenticação para realizar os testes, praticamente vamos reutilizar o módulo `jwt-simple` para criar um token válido que será usado no cabeçalho de todos os testes. Esse token será gerado repetidamente dentro do callback da função `beforeEach(done)`. Mas, para gerá-lo, antes teremos de excluir todos os usuários por meio da função `Users.destroy({where: {}})` para, em seguida, criar um novo e único usuário na base via função `Users.create()`.

Faremos o mesmo fluxo para criação de tarefas, porém no lugar da função `Tasks.create()`, será usada a função `Tasks.bulkCreate()`, que permite enviar um array de várias tarefas a serem inseridas em uma única execução (essa função é muito útil para inclusão em lote de dados).

As tarefas utilizarão o `user.id` do usuário, criado para garantir que elas são do usuário autenticado. Na reta final, pegamos a primeira tarefa criada por meio do trecho `fakeTask = tasks[0]` para reutilizar seu `id` nos testes que necessitam de um `id` de tarefa como parâmetro na rota. Também geramos um token válido através da função `jwt.encode({id: user.id}, jwtSecret)`.

Ambos os objetos `fakeTask` e `token` são criados em um escopo acima da função `beforeEach(done)`, para que sejam reutilizados nos testes. Para entender em detalhes, faça a seguinte implementação:

```
beforeEach(done => {
  Users
    .destroy({where: {}})
    .then(() => Users.create({
      name: "John",
      email: "john@mail.net",
```

```
      password: "12345"
    }))
    .then(user => {
      Tasks
        .destroy({where: {}})
        .then(() => Tasks.bulkCreate([{
          id: 1,
          title: "Work",
          user_id: user.id
        }, {
          id: 2,
          title: "Study",
          user_id: user.id
        }]))
        .then(tasks => {
          fakeTask = tasks[0];
          token = jwt.encode({id: user.id}, jwtSecret);
          done();
        });
    });
});
```

Com as rotinas de pré-testes pronta, vamos codificar todos os testes dos endpoints de tarefas, começando com o teste para a rota `GET /tasks`. Nele, é realizada uma requisição via função `request.get("/tasks")`, usando também a função `set("Authorization", JWT ${token})`, que permite enviar um cabeçalho na requisição, que neste caso, é enviado o cabeçalho `Authorization` com o valor do token de autenticação.

Para garantir que o teste seja realizado com sucesso:

1. Checamos o `status 200` via função `expect(200)`;
2. Aplicamos uma simples validação para garantir que será retornado um array de tamanho 2 via função `expect(res.body).to.have.length(2)`;
3. Comparamos se os títulos das 2 tarefas são iguais as que foram criadas pela função `Tasks.bulkCreate()`.

```
describe("GET /tasks", () => {
  describe("status 200", () => {
    it("returns a list of tasks", done => {
      request.get("/tasks")
        .set("Authorization", `JWT ${token}`)
        .expect(200)
        .end((err, res) => {
          expect(res.body).to.have.length(2);
          expect(res.body[0].title).to.eql("Work");
          expect(res.body[1].title).to.eql("Study");
          done(err);
        });
    });
  });
});
```

Para testar o caso de sucesso da rota `POST /tasks`, não há segredos: basicamente informamos o cabeçalho com token de autenticação e um título para uma nova tarefa. Como saída, testamos se a resposta retorna `status 200`, e se o objeto `req.body` possui o mesmo título que foi enviado para cadastrar essa nova tarefa.

```
describe("POST /tasks", () => {
  describe("status 200", () => {
    it("creates a new task", done => {
      request.post("/tasks")
        .set("Authorization", `JWT ${token}`)
        .send({title: "Run"})
        .expect(200)
        .end((err, res) => {
          expect(res.body.title).to.eql("Run");
          expect(res.body.done).to.be.false;
          done(err);
        });
    });
  });
});
```

Agora vamos testar 2 simples fluxos da rota `GET /tasks/:id`. No caso de sucesso, usaremos o `id` do objeto `fakeTask` para

garantir que retorne uma tarefa válida. Para testar o comportamento quando é informado um `id` de tarefa inválido, vamos utilizar a função `expect(404)` para testar o `status 404`, que indica que a requisição não encontrou um recurso.

```
describe("GET /tasks/:id", () => {
  describe("status 200", () => {
    it("returns one task", done => {
      request.get(`/tasks/${fakeTask.id}`)
        .set("Authorization", `JWT ${token}`)
        .expect(200)
        .end((err, res) => {
          expect(res.body.title).to.eql("Work");
          done(err);
        });
    });
  });
  describe("status 404", () => {
    it("throws error when task not exist", done => {
      request.get("/tasks/0")
        .set("Authorization", `JWT ${token}`)
        .expect(404)
        .end((err, res) => done(err));
    });
  });
});
```

Para finalizar os testes, vamos testar apenas o comportamento de sucesso das rotas `PUT /tasks/:id` e `DELETE /tasks/:id`. Ambos usarão praticamente as mesmas funções, exceto que um teste executará a função `request.put()` e o outro, `request.delete()`. Porém, ambos vão esperar que o sucesso da requisição retorne um `status 204` através da função `expect(204)`.

```
describe("PUT /tasks/:id", () => {
  describe("status 204", () => {
    it("updates a task", done => {
      request.put(`/tasks/${fakeTask.id}`)
        .set("Authorization", `JWT ${token}`)
```

```
        .send({
          title: "Travel",
          done: true
        })
        .expect(204)
        .end((err, res) => done(err));
    });
  });
});
describe("DELETE /tasks/:id", () => {
  describe("status 204", () => {
    it("removes a task", done => {
      request.delete(`/tasks/${fakeTask.id}`)
        .set("Authorization", `JWT ${token}`)
        .expect(204)
        .end((err, res) => done(err));
    });
  });
});
```

Parabéns! Acabamos os testes do recurso tarefas. Caso você execute novamente o comando `npm test`, você terá o seguinte resultado:

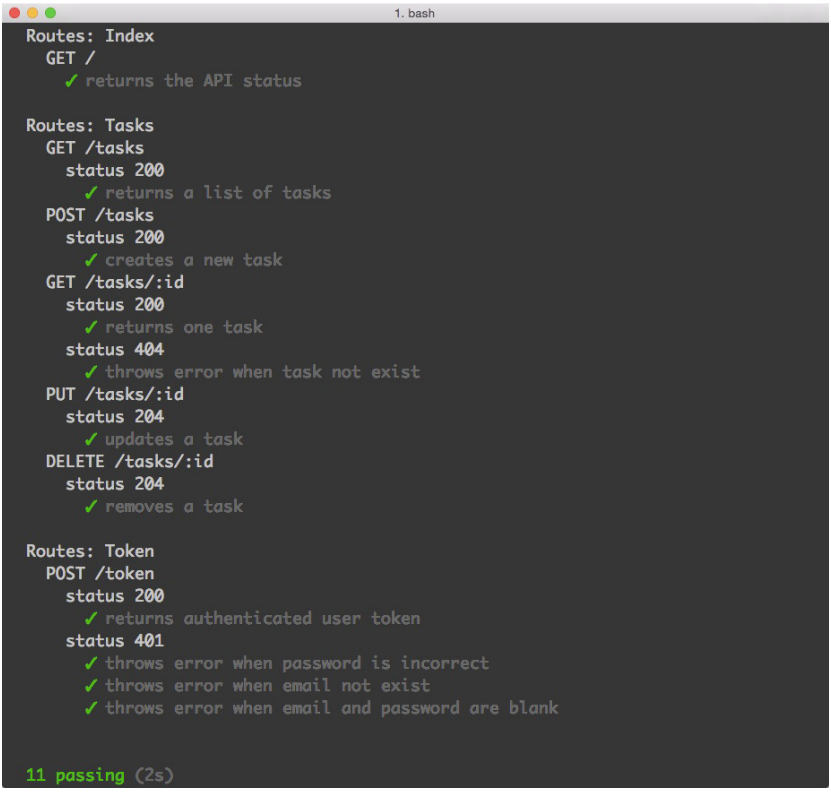


Figura 9.1: Testando endpoints de tarefas

9.2 TESTANDO OS ENDPOINTS DE USUÁRIO

Para testar o recurso de gestão de usuários, é mais simples ainda, pois praticamente vamos utilizar tudo o que já foi explicado nos testes anteriores. Para começar, crie o arquivo `test/routes/users.js` com a seguinte estrutura:

```
import jwt from "jwt-simple";

describe("Routes: Tasks", () => {
  const Users = app.db.models.Users;
```

```
  const jwtSecret = app.libs.config.jwtSecret;
  let token;
  beforeEach(done => {
    // Código de teste
  });
  describe("GET /user", () => {
    describe("status 200", () => {
      it("returns an authenticated user", done => {
        // Código de teste
      });
    });
  });
  describe("DELETE /user", () => {
    describe("status 200", () => {
      it("deletes an authenticated user", done => {
        // Código de teste
      });
    });
  });
  describe("POST /users", () => {
    describe("status 200", () => {
      it("creates a new user", done => {
        // Código de teste
      });
    });
  });
});
```

A lógica de pré-testes será mais simplificada, porém terá também a geração de um token de autenticação válido. Veja a seguir como implementar a função `beforeEach(done)`:

```
beforeEach(done => {
  Users
    .destroy({where: {}})
    .then(() => Users.create({
      name: "John",
      email: "john@mail.net",
      password: "12345"
    }))
    .then(user => {
      token = jwt.encode({id: user.id}, jwtSecret);
      done();
```

```
    });
});
```

Agora, para implementar os testes, vamos começar testando a requisição `GET /user`, que retorna os dados de um usuário autenticado, que basicamente envia um token de autenticação e recebe como resposta os dados do usuário que foi criado na função `beforeEach(done)`.

```
describe("GET /user", () => {
  describe("status 200", () => {
    it("returns an authenticated user", done => {
      request.get("/user")
        .set("Authorization", `JWT ${token}`)
        .expect(200)
        .end((err, res) => {
          expect(res.body.name).to.eql("John");
          expect(res.body.email).to.eql("john@mail.net");
          done(err);
        });
    });
  });
});
```

Em seguida, codificaremos os testes para a rota `DELETE /user`, para testar se a exclusão de usuário autenticado. Os testes para esse caso são mais simples: enviar um token e esperar como sucesso o `status 204`.

```
describe("DELETE /user", () => {
  describe("status 200", () => {
    it("deletes an authenticated user", done => {
      request.delete("/user")
        .set("Authorization", `JWT ${token}`)
        .expect(204)
        .end((err, res) => done(err));
    });
  });
});
```

Para finalizar, vamos implementar o teste mais simples que faz

um cadastro de novo usuário na API. Este não exige token, afinal, é uma rota aberta para novos usuários cadastrarem uma conta na aplicação. Veja a seguir o código desse teste:

```
describe("POST /users", () => {
  describe("status 200", () => {
    it("creates a new user", done => {
      request.post("/users")
        .send({
          name: "Mary",
          email: "mary@mail.net",
          password: "12345"
        })
        .expect(200)
        .end((err, res) => {
          expect(res.body.name).to.eql("Mary");
          expect(res.body.email).to.eql("mary@mail.net");
          done(err);
        });
    });
  });
});
```

Para testar, execute o comando `npm test`. Se tudo rodar com sucesso, você terá um lindo report semelhante a este:

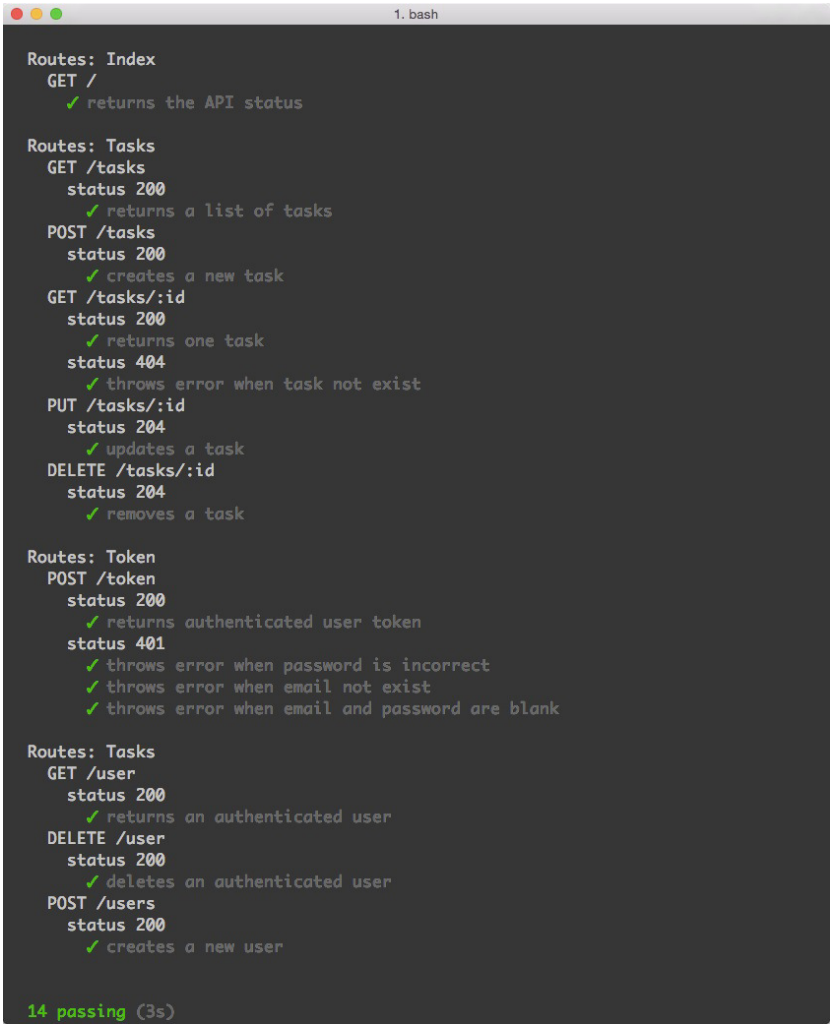


Figura 9.2: Testando endpoints de usuário

Conclusão

Se você chegou até esta etapa, então você já desenvolveu uma pequena, porém poderosa, API, utilizando Node.js e banco de

dados do tipo SQL. Tudo isso já funcionando com o mínimo de testes para garantir a qualidade de código no projeto.

No próximo capítulo, vamos usar uma ferramenta muito útil para geração de documentação de APIs. Continue a leitura que ainda tem muita coisa legal para explorarmos!

CAPÍTULO 10

DOCUMENTANDO UMA API

Se você chegou até este capítulo e sua aplicação está funcionando corretamente – com rotas para os recursos de gestão de tarefas e usuários, integrados ao banco de dados, e com autenticação de usuários através do JSON Web Token –, meus parabéns! Você criou, seguindo boas práticas, uma API Rest utilizando Node.js. Se você pretende usar esse projeto piloto como base para construir sua própria API, então você já tem uma aplicação pronta para enviá-la para um servidor de ambiente de produção.

10.1 INTRODUÇÃO A FERRAMENTA APIDOC

Neste capítulo, aprenderemos como documentar os endpoints de uma API, afinal, é uma boa prática disponibilizar uma documentação sobre como as aplicações clientes poderão se autenticar e consumir os dados de uma API. O mais legal é que vamos utilizar uma ferramenta muito simples de usar, e toda a documentação da nossa aplicação será feita por meio de comentários padronizados dentro dos códigos das rotas.

Usaremos a ferramenta **apiDoc**, um módulo Node.js que,

através da leitura de seus comentários padronizados, ele consegue gerar uma documentação bonita e elegante para APIs.

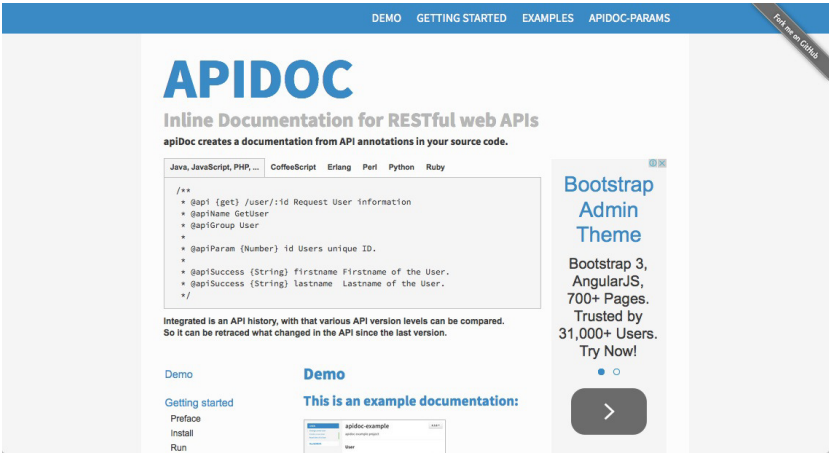


Figura 10.1: Homepage do apiDoc

Esse módulo é um CLI (::Command Line Interface::), e é recomendável instalá-lo como módulo global (através do comando `npm install -g`). Porém, no nosso caso, vamos criar um comando `npm` para usá-lo toda vez que iniciarmos o servidor da API. Logo, sua instalação será como um módulo local, semelhante aos demais que já foram instalados.

Instale-o pelo comando:

```
npm install apidoc@0.15.1 --save-dev
```

Como o objeto atualiza a documentação toda vez que iniciarmos o servidor, então vamos modificar o comando `npm start`. Primeiro, vamos criar o novo comando `npm run apidoc`, que executará o comando `apidoc -i routes/ -o public/apidoc`. Depois, modificaremos o atributo

`scripts.start` para que ele gere a documentação da API e, em seguida, inicie o servidor da aplicação. Também incluiremos o atributo `apidoc.name`, que será o título da página de documentação da API e a propriedade `apidoc.template.forceLanguage` para definir o idioma padrão da página de documentaçao da API.

Abra e edite o `package.json`, fazendo a seguinte alteração:

```
{
  "name": "ntask-api",
  "version": "1.0.0",
  "description": "API de gestão de tarefas",
  "main": "index.js",
  "scripts": {
    "start": "npm run apidoc && babel-node index.js",
    "apidoc": "apidoc -i routes/ -o public/apidoc",
    "test": "NODE_ENV=test mocha test/**/*.js"
  },
  "apidoc": {
    "name": "Documentação - Node Task API",
    "template": {
      "forceLanguage": "pt_br"
    }
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babel-cli": "^6.5.1",
    "babel-preset-es2015": "^6.5.0",
    "bcrypt": "^0.8.5",
    "body-parser": "^1.15.0",
    "consign": "^0.1.2",
    "express": "^4.13.4",
    "jwt-simple": "^0.4.1",
    "passport": "^0.3.2",
    "passport-jwt": "^2.0.0",
    "sequelize": "^3.19.2",
    "sqlite3": "^3.1.1"
  },
  "devDependencies": {
    "apidoc": "^0.15.1",
    "babel-register": "^6.5.2",
```

```
    "chai": "^3.5.0",
    "mocha": "^2.4.5",
    "supertest": "^1.2.0"
  }
}
```

A partir de agora, toda vez que você executar o comando `npm start`, se você quiser apenas gerar uma nova documentação sem iniciar o servidor, você pode rodar apenas `npm run apidoc`. Ambos os comandos vão varrer e procurar todos os comentários existentes no diretório `routes` para gerar a documentação da API, que será salva na pasta `public/apidoc` e, em seguida, iniciará o servidor.

Para que seja possível visualizar a página de documentação, primeiro teremos de habilitar o servidor de arquivos estáticos do Express, para que ele sirva todo o conteúdo estático existente na pasta `public`. Para habilitá-lo, basta incluir o middleware `app.use(express.static("public"))` no final do arquivo `libs/middlewares.js`. Veja como fica:

```
import bodyParser from "body-parser";
import express from "express";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

Para validar se está tudo funcionando, vamos documentar, por enquanto, o endpoint de status da API – o endpoint `/` –, e vamos

usar os seguintes comentários:

- `@api` : informa o tipo, endereço e título do endpoint;
- `@apiGroup` : informa o nome do grupo de endpoints;
- `@apiSuccess` : descreve os campos e seus tipos de dados em uma resposta de sucesso;
- `@apiSuccessExample` : apresenta um exemplo de resposta de sucesso.

Para documentar esse endpoint, edite o arquivo `routes/index.js` com o seguinte código:

```
module.exports = app => {
  /**
   * @api {get} / API Status
   * @apiGroup Status
   * @apiSuccess {String} status Mensagem de status da API
   * @apiSuccessExample {json} Sucesso
   *    HTTP/1.1 200 OK
   *    {"status": "NTask API"}
   */
  app.get("/", (req, res) => {
    res.json({status: "NTask API"});
  });
};
```

Para testar essas alterações, basta reiniciar seu servidor por meio do comando `npm start` e, em seguida, acessar no browser o endereço: http://localhost:3000/apidoc.

Se não ocorrer erros, você visualizará uma linda página de documentação de APIs.

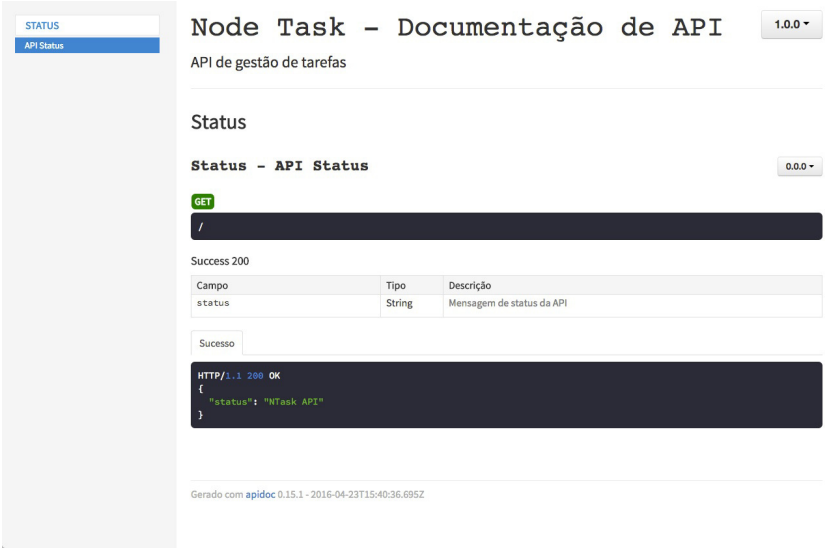


Figura 10.2: Documentação de Status da API

10.2 DOCUMENTANDO A GERAÇÃO DE TOKENS

Agora, vamos explorar mais a fundo as funcionalidades do apiDoc, documentando as restantes rotas da API.

Para iniciar, vamos documentar a rota `/token`. Ela possui alguns detalhes extras para ser documentados. Nela, não só usaremos os itens explicados na seção anterior como também utilizaremos esses novos itens:

- `@apiParam`: descreve um parâmetro de entrada, que pode ser ou não obrigatório o seu envio em uma requisição;
- `@apiParamExample`: apresenta um exemplo real de parâmetros de entrada, no nosso caso, vamos exibir um JSON de entrada;

- `@apiErrorExample` : mostra um exemplo de erro que a API pode gerar se não forem enviado os parâmetros corretamente.

Para entender na prática o uso desses novos itens, edite o `routes/token.js` , seguindo os comentários a seguir:

```
import jwt from "jwt-simple";
module.exports = app => {
  const cfg = app.libs.config;
  const Users = app.db.models.Users;
  /**
   * @api {post} /token Token autenticado
   * @apiGroup Credencial
   * @apiParam {String} email Email de usuário
   * @apiParam {String} password Senha de usuário
   * @apiParamExample {json} Entrada
   *    {
   *      "email": "john@connor.net",
   *      "password": "123456"
   *    }
   * @apiSuccess {String} token Token de usuário autenticado
   * @apiSuccessExample {json} Sucesso
   *    HTTP/1.1 200 OK
   *    {"token": "xyz.abc.123.hgf"}
   * @apiErrorExample {json} Erro de autenticação
   *    HTTP/1.1 401 Unauthorized
   */
  app.post("/token", (req, res) => {
    // Lógica do /token que foi explicada no capítulo 7
  });
};
```

10.3 DOCUMENTANDO RECURSO DE GESTÃO DE USUÁRIOS

Nesta e na próxima seção, vamos documentar os 2 recursos principais da API: usuários e tarefas. Como a maioria das rotas desses recursos necessita de um **Token** de usuário autenticado –

que é enviado pelo ::header:: da requisição –, vamos usar os seguintes itens para descrever seus parâmetros:

- `@apiHeader` : descreve nome e tipo de dado de um header;
- `@apiHeaderExample` : exibe um exemplo de header a ser usado na requisição.

Abra o `routes/users.js` e vamos começar documentando a rota `GET /user` .

```
module.exports = app => {
  const Users = app.db.models.Users;
  app.route("/user")
    .all(app.auth.authenticate())
    /**
     * @api {get} /user Exibe usuário autenticado
     * @apiGroup Usuário
     * @apiHeader {String} Authorization Token de usuário
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiSuccess {Number} id Id de registro
     * @apiSuccess {String} name Nome
     * @apiSuccess {String} email Email
     * @apiSuccessExample {json} Sucesso
     *    HTTP/1.1 200 OK
     *    {
     *      "id": 1,
     *      "name": "John Connor",
     *      "email": "john@connor.net"
     *    }
     * @apiErrorExample {json} Erro de consulta
     *    HTTP/1.1 412 Precondition Failed
     */
    .get((req, res) => {
      // Lógica explicada no capítulo 7
    })
    // continuação das rotas DELETE e POST
```

Em seguida, vamos documentar a rota `DELETE /user` :

```
    /**
```

```
    * @api {delete} /user Exclui usuário autenticado
    * @apiGroup Usuário
    * @apiHeader {String} Authorization Token de usuário
    * @apiHeaderExample {json} Header
    *    {"Authorization": "JWT xyz.abc.123.hgf"}
    * @apiSuccessExample {json} Sucesso
    *    HTTP/1.1 204 No Content
    * @apiErrorExample {json} Erro na exclusão
    *    HTTP/1.1 412 Precondition Failed
    */
  .delete((req, res) => {
   // Lógica explicada no capítulo 7
  })
  // continuação da rota POST
```

Para finalizar, ainda no mesmo arquivo `routes/users.js`, documentaremos sua última rota, a `POST /user`, usando vários itens para descrever todos os seus campos de entrada e saída:

```
/**
 * @api {post} /users Cadastra novo usuário
 * @apiGroup Usuário
 * @apiParam {String} name Nome
 * @apiParam {String} email Email
 * @apiParam {String} password Senha
 * @apiParamExample {json} Entrada
 *    {
 *      "name": "John Connor",
 *      "email": "john@connor.net",
 *      "password": "123456"
 *    }
 * @apiSuccess {Number} id Id de registro
 * @apiSuccess {String} name Nome
 * @apiSuccess {String} email Email
 * @apiSuccess {String} password Senha criptografada
 * @apiSuccess {Date} updated_at Data de atualização
 * @apiSuccess {Date} created_at Data de cadastro
 * @apiSuccessExample {json} Sucesso
 *    HTTP/1.1 200 OK
 *    {
 *      "id": 1,
 *      "name": "John Connor",
 *      "email": "john@connor.net",
```

```
   *      "password": "$2a$10$SK1B1",
   *      "updated_at": "2015-09-24T15:46:51.778Z",
   *      "created_at": "2015-09-24T15:46:51.778Z"
   *    }
   * @apiErrorExample {json} Erro no cadastro
   *    HTTP/1.1 412 Precondition Failed
   */
  app.post("/users", (req, res) => {
    // Lógica explicada no capítulo 7
  });
};
```

10.4 DOCUMENTANDO RECURSO DE GESTÃO DE TAREFAS

Dando continuidade à nossa documentação de API, vamos agora finalizar essa tarefa documentando os endpoints do arquivo `routes/tasks.js`, e descrevendo inicialmente a rota `GET /tasks`:

```
module.exports = app => {
  const Tasks = app.db.models.Tasks;
  app.route("/tasks")
    .all(app.auth.authenticate())
    /**
     * @api {get} /tasks Lista tarefas
     * @apiGroup Tarefas
     * @apiHeader {String} Authorization Token de usuário
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiSuccess {Object[]} tasks Lista de tarefas
     * @apiSuccess {Number} tasks.id Id de registro
     * @apiSuccess {String} tasks.title Título da tarefa
     * @apiSuccess {Boolean} tasks.done Tarefa foi concluída?
     * @apiSuccess {Date} tasks.updated_at Data de atualização
     * @apiSuccess {Date} tasks.created_at Data de cadastro
     * @apiSuccess {Number} tasks.user_id Id do usuário
     * @apiSuccessExample {json} Sucesso
     *    HTTP/1.1 200 OK
     *    [{
     *      "id": 1,
```

```
 *      "title": "Estudar",
 *      "done": false
 *      "updated_at": "2015-09-24T15:46:51.778Z",
 *      "created_at": "2015-09-24T15:46:51.778Z",
 *      "user_id": 1
 *    }]
 * @apiErrorExample {json} Erro de consulta
 *    HTTP/1.1 412 Precondition Failed
 */
.get((req, res) => {
 // Lógica implementada no capítulo 7
})
```

Em seguida, documentaremos a rota `POST /tasks`:

```
/**
 * @api {post} /tasks Cadastra uma tarefas
 * @apiGroup Tarefas
 * @apiHeader {String} Authorization Token de usuário
 * @apiHeaderExample {json} Header
 *    {"Authorization": "JWT xyz.abc.123.hgf"}
 * @apiParam {String} title Título da tarefa
 * @apiParamExample {json} Entrada
 *    {"title": "Estudar"}
 * @apiSuccess {Number} id Id de registro
 * @apiSuccess {String} title Título da tarefa
 * @apiSuccess {Boolean} done=false Tarefa foi concluída?
 * @apiSuccess {Date} updated_at Data de atualização
 * @apiSuccess {Date} created_at Data de cadastro
 * @apiSuccess {Number} user_id Id do usuário
 * @apiSuccessExample {json} Sucesso
 *    HTTP/1.1 200 OK
 *    {
 *      "id": 1,
 *      "title": "Estudar",
 *      "done": false,
 *      "updated_at": "2015-09-24T15:46:51.778Z",
 *      "created_at": "2015-09-24T15:46:51.778Z",
 *      "user_id": 1
 *    }
 * @apiErrorExample {json} Erro de consulta
 *    HTTP/1.1 412 Precondition Failed
 */
.post((req, res) => {
```

```
 // Lógica implementada no capítulo 7
 })
```

Depois, vamos documentar a rota `GET /tasks/:id` , com os seguintes comentários:

```
/**
 * @api {get} /tasks/:id Exibe uma tarefa
 * @apiGroup Tarefas
 * @apiHeader {String} Authorization Token de usuário
 * @apiHeaderExample {json} Header
 *    {"Authorization": "JWT xyz.abc.123.hgf"}
 * @apiParam {id} id Id da tarefa
 * @apiSuccess {Number} id Id de registro
 * @apiSuccess {String} title Título da tarefa
 * @apiSuccess {Boolean} done Tarefa foi concluída?
 * @apiSuccess {Date} updated_at Data de atualização
 * @apiSuccess {Date} created_at Data de cadastro
 * @apiSuccess {Number} user_id Id do usuário
 * @apiSuccessExample {json} Sucesso
 *    HTTP/1.1 200 OK
 *    {
 *      "id": 1,
 *      "title": "Estudar",
 *      "done": false
 *      "updated_at": "2015-09-24T15:46:51.778Z",
 *      "created_at": "2015-09-24T15:46:51.778Z",
 *      "user_id": 1
 *    }
 * @apiErrorExample {json} Tarefa não existe
 *    HTTP/1.1 404 Not Found
 * @apiErrorExample {json} Erro de consulta
 *    HTTP/1.1 412 Precondition Failed
 */
 .get((req, res) => {
  // Lógica implementada no capítulo 7
 })
```

Agora, a `PUT /tasks/:id` :

```
/**
 * @api {put} /tasks/:id Atualiza uma tarefa
 * @apiGroup Tarefas
 * @apiHeader {String} Authorization Token de usuário
```

```
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiParam {id} id Id da tarefa
     * @apiParam {String} title Título da tarefa
     * @apiParam {Boolean} done Tarefa foi concluída?
     * @apiParamExample {json} Entrada
     *    {
     *      "title": "Trabalhar",
     *      "done": true
     *    }
     * @apiSuccessExample {json} Sucesso
     *    HTTP/1.1 204 No Content
     * @apiErrorExample {json} Erro de consulta
     *    HTTP/1.1 412 Precondition Failed
     */
     .put((req, res) => {
      // Lógica implementada no capítulo 7
     })
```

Por último, vamos finalizar este capítulo documentando a rota `DELETE /tasks/:id`:

```
    /**
     * @api {delete} /tasks/:id Exclui uma tarefa
     * @apiGroup Tarefas
     * @apiHeader {String} Authorization Token de usuário
     * @apiHeaderExample {json} Header
     *    {"Authorization": "JWT xyz.abc.123.hgf"}
     * @apiParam {id} id Id da tarefa
     * @apiSuccessExample {json} Sucesso
     *    HTTP/1.1 204 No Content
     * @apiErrorExample {json} Erro de consulta
     *    HTTP/1.1 412 Precondition Failed
     */
     .delete((req, res) => {
      // Lógica implementada no capítulo 7
     });
};
```

Vamos testar? Basta reiniciar o servidor e depois acesse o endereço: http://localhost:3000/apidoc.

Dessa vez, temos uma página de documentação completa que

descreve bem o passo a passo para um novo desenvolvedor criar uma aplicação cliente, para consumir nossa API.

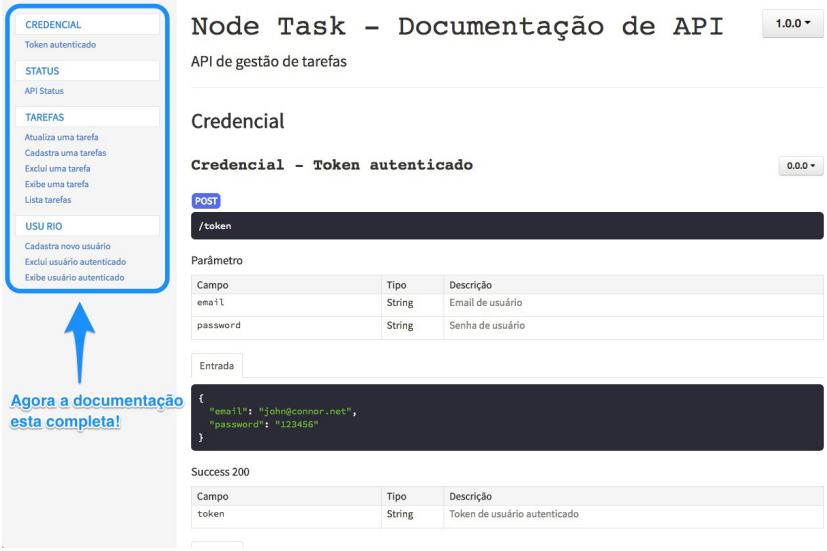


Figura 10.3: Agora, a documentação da API está completa!

Conclusão

Parabéns! Acabamos mais um excelente capítulo. Agora não só temos uma API funcional como também uma documentação completa para permitir que outros desenvolvedores criem aplicações client-side utilizando nossa API.

Continue lendo, pois, no próximo episódio, vamos incluir alguns frameworks e boas práticas para que nossa API trabalhe em ambiente de produção corretamente.

Capítulo 11

PREPARANDO O AMBIENTE DE PRODUÇÃO

11.1 INTRODUÇÃO AO CORS

Caso você não saiba, o CORS (*Cross-origin resource sharing*) é um mecanismo muito importante do HTTP. Ele é responsável por permitir ou barrar requisições assíncronas que são realizadas por outros domínios.

O CORS, na prática, são apenas headers do HTTP que são incluídos no server-side da aplicação. Tais headers podem informar qual domínio poderá consumir a API, quais métodos do HTTP serão permitidos e, principalmente, quais endpoints serão compartilhados de forma pública para outros domínios de outras aplicações consumirem.

11.2 HABILITANDO CORS NA API

Como estamos desenvolvendo uma API que servirá dados para qualquer tipo de aplicação cliente, então teremos de habilitar o CORS como middleware global, para que todos endpoints sejam públicos. Ou seja, para que qualquer cliente possa realizar requisições em nossa API. Para habilitar o CORS na API, vamos

instalar e usar o módulo `cors`:

```
npm install cors@2.7.1 --save
```

Em seguida, vamos iniciá-lo via função `app.use(cors())` no arquivo de middlewares, o `libs/middlewares.js`:

```
import bodyParser from "body-parser";
import express from "express";
import cors from "cors";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(cors());
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

Ao usar somente a função `cors()`, estaremos liberando acesso completo de nossa API para qualquer cliente consumir. Porém, o recomendado é ter controle de quais domínios clientes vão acessá-la, quais métodos vão utilizar e, principalmente, quais headers serão obrigatórios para o cliente informar no momento da requisição. No nosso caso, vamos configurar apenas três atributos: `origin` (domínios permitidos), `methods` (métodos permitidos) e `allowedHeaders` (headers obrigatórios).

Ainda no `libs/middlewares.js`, modifique a função `app.use(cors())` por esta:

```
import bodyParser from "body-parser";
import express from "express";
import cors from "cors";
```

```
module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

Agora temos uma API que vai aceitar somente aplicações clientes do domínio origem: http://localhost:3001.

Fique tranquilo, pois esse domínio será a nossa aplicação cliente que vamos construir em detalhes no próximo capítulo!

> **UM POUCO MAIS SOBRE CORS**
>
> Para você estudar mais a fundo sobre o CORS, para entender seus *headers* e, principalmente, como criar uma regra mais customizada para sua API, recomendo que acesse https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.

11.3 GERANDO LOGS DE REQUISIÇÕES

Nesta seção, vamos configurar nossa aplicação para que ela

reporte e gere arquivo de logs das requisições realizadas. Para isso, usaremos o módulo `winston`, que é especializado em tratar diversos tipos de logs.

No nosso caso, os logs de requisições serão tratados por meio do módulo `morgan`, que é um middleware responsável por gerar logs das requisições no servidor. Também vamos tratar os logs de comandos SQLs gerados no banco de dados. Primeiro, instale os módulos `winston` e `morgan`:

```
npm install winston@2.1.1 morgan@1.6.1 --save
```

Feito isso, vamos implementar um código para configurar e carregar o `winston`. Nele, verificaremos se existe a pasta `logs`, usando o módulo nativo `fs` (*File System*). Em seguida, será implementada uma simples condicional via função `fs.existsSync("logs")`, para checar se existe ou não a pasta `logs`. Se essa pasta não existir, ela será criada pela função `fs.mkdirSync("logs")`.

Depois dessa verificação da existência da pasta `logs`, basta instanciar e exportar o objeto `module.exports = new winston.Logger`. Como vamos gerar arquivos de logs, o nosso objeto de logs usará como `transports` o objeto `new winston.transports.File`, que é responsável por criar e manter vários arquivos de logs recentes. Crie o arquivo `libs/logger.js`, da seguinte maneira:

```
import fs from "fs";
import winston from "winston";

if (!fs.existsSync("logs")) {
  fs.mkdirSync("logs");
}
```

```
module.exports = new winston.Logger({
  transports: [
    new winston.transports.File({
      level: "info",
      filename: "logs/app.log",
      maxsize: 1048576,
      maxFiles: 10,
      colorize: false
    })
  ]
});
```

Agora, vamos utilizar nosso `libs/logger.js` em dois pontos importantes de nossa aplicação. Primeiro, usaremos para gerar logs dos comandos SQLs. Vamos modificar o arquivo `libs/config.development.js` para ele carregar nosso módulo `logger`. Vamos usar sua função `logger.info()` como callback do atributo `logging` do Sequelize para capturarmos cada comando SQL gerado na aplicação. Para fazer isso, edite o `libs/config.development.js` da seguinte maneira:

```
import logger from "./logger.js";

module.exports = {
  database: "ntask",
  username: "",
  password: "",
  params: {
    dialect: "sqlite",
    storage: "ntask.sqlite",
    logging: (sql) => {
      logger.info(`[${new Date()}] ${sql}`);
    },
    define: {
      underscored: true
    }
  },
  jwtSecret: "Nta$K-AP1",
  jwtSession: {session: false}
};
```

Para finalizar, vamos utilizar o módulo `logger` que criamos para gerar logs das requisições feitas em nosso servidor. Para isso, usaremos o módulo `morgan` e incluiremos no topo dos middlewares a função `app.use(morgan("common"))`, para permitir a geração de logs das requisições.

Para enviarmos esses logs para o nosso módulo `logger`, basta adicionar o atributo `stream` com uma função callback chamada `write(message)` e, em seguida, enviar a variável `message` para nossa função de log, a `logger.info(message)`. Para entender melhor essa implementação, edite o arquivo `libs/middlewares.js` da seguinte maneira:

```
import bodyParser from "body-parser";
import express from "express";
import morgan from "morgan";
import cors from "cors";
import logger from "./logger.js";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(morgan("common", {
    stream: {
      write: (message) => {
        logger.info(message);
      }
    }
  }));
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
```

```
  app.use(express.static("public"));
};
```

Para testar a geração de logs, basta reiniciar o servidor e acessar várias vezes qualquer endereço da API, por exemplo o http://localhost:3000/.

Após realizar algumas requisições na API, acesse o diretório `logs` da raiz do projeto. Lá com certeza terá um arquivo de logs com dados de requisições semelhantes a este:

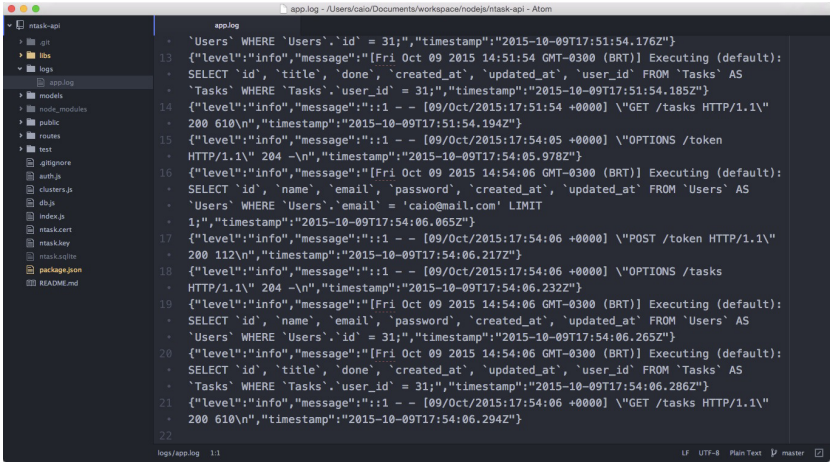

app.log - /Users/caio/Documents/workspace/nodejs/ntask-api - Atom
ntask-api
app.log
`Users` WHERE `Users`.`id` = 31;","timestamp":"2015-10-09T17:51:54.176Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:51:54 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `title`, `done`, `created_at`, `updated_at`, `user_id` FROM `Tasks` AS
`Tasks` WHERE `Tasks`.`user_id` = 31;","timestamp":"2015-10-09T17:51:54.185Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:51:54 +0000] \"GET /tasks HTTP/1.1\"
200 610\n","timestamp":"2015-10-09T17:51:54.194Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:05 +0000] \"OPTIONS /token
HTTP/1.1\" 204 -\n","timestamp":"2015-10-09T17:54:05.978Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:54:06 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `name`, `email`, `password`, `created_at`, `updated_at` FROM `Users` AS
`Users` WHERE `Users`.`email` = 'caio@mail.com' LIMIT
1;","timestamp":"2015-10-09T17:54:06.065Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:06 +0000] \"POST /token HTTP/1.1\"
200 112\n","timestamp":"2015-10-09T17:54:06.217Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:06 +0000] \"OPTIONS /tasks
HTTP/1.1\" 204 -\n","timestamp":"2015-10-09T17:54:06.232Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:54:06 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `name`, `email`, `password`, `created_at`, `updated_at` FROM `Users` AS
`Users` WHERE `Users`.`id` = 31;","timestamp":"2015-10-09T17:54:06.265Z"}
{"level":"info","message":"[Fri Oct 09 2015 14:54:06 GMT-0300 (BRT)] Executing (default):
SELECT `id`, `title`, `done`, `created_at`, `updated_at`, `user_id` FROM `Tasks` AS
`Tasks` WHERE `Tasks`.`user_id` = 31;","timestamp":"2015-10-09T17:54:06.286Z"}
{"level":"info","message":"::1 - - [09/Oct/2015:17:54:06 +0000] \"GET /tasks HTTP/1.1\"
200 610\n","timestamp":"2015-10-09T17:54:06.294Z"}

Figura 11.1: Logs das requisições

11.4 CONFIGURANDO PROCESSAMENTO PARALELO COM MÓDULO CLUSTER

Infelizmente, o Node.js não trabalha com *threads*. Isso é algo que, na opinião de alguns desenvolvedores, é considerado um ponto negativo, e que provoca um certo desinteresse em aprender ou levar a sério essa tecnologia. Entretanto, apesar de ele ser *single-thread*, é possível, sim, prepará-lo para trabalhar com

processamento paralelo. Para isso, existe nativamente um módulo chamado `cluster`.

Ele basicamente instancia novos processos de uma aplicação, trabalhando de forma distribuída e, quando trabalhamos com uma aplicação web, esse módulo se encarrega de compartilhar a mesma porta da rede entre os *clusters* ativos. O número de processos a serem criados é você quem determina, e é claro que a boa prática é instanciar um total de processos relativo à quantidade de núcleos do processador do servidor, ou também uma quantidade relativa a **núcleos X processadores**.

Por exemplo, se tenho um único processador de oito núcleos, então, posso instanciar oito processos, criando assim uma rede de oito *clusters*. Mas, caso tenha quatro processadores de oito núcleos cada, é possível criar uma rede de trinta e dois *clusters* em ação.

Para garantir que os *clusters* trabalhem de forma distribuída e organizada, é necessário que exista um processo pai, mais conhecido como *cluster master*. Ele é o responsável por balancear a carga de processamento entre os demais *clusters*, distribuindo-a para os processos filhos, que são chamados de *cluster slave*. Implementar essa técnica no Node.js é muito simples, visto que toda a distribuição de processamento é executada de forma abstraída para o desenvolvedor.

Outra vantagem é que os *clusters* são independentes uns dos outros. Ou seja, caso um *cluster* saia do ar, os demais continuarão servindo a aplicação mantendo o sistema no ar. Porém, é necessário gerenciar as instâncias e encerramento desses *clusters* manualmente para garantir o retorno do *cluster* que saiu do ar.

Com base nesses conceitos, vamos aplicar na prática a implementação de *clusters*. Crie no diretório raiz o arquivo `clusters.js`, para que, por meio dele, seja carregado clusters da nossa aplicação. Veja o código a seguir:

```
import cluster from "cluster";
import os from "os";

const CPUS = os.cpus();
if (cluster.isMaster) {
  CPUS.forEach(() => cluster.fork());
  cluster.on("listening", worker => {
    console.log("Cluster %d conectado", worker.process.pid);
  });
  cluster.on("disconnect", worker => {
    console.log("Cluster %d desconectado", worker.process.pid);
  });
  cluster.on("exit", worker => {
    console.log("Cluster %d saiu do ar", worker.process.pid);
    cluster.fork();
    // Garante que um novo cluster inicie se um antigo morrer
  });
} else {
  require("./index.js");
}
```

Dessa vez, para levantar o servidor, primeiro edite o `package.json` dentro do atributo `scripts`, para criar o comando `npm run clusters`, conforme o código a seguir:

```
"scripts": {
  "start": "npm run apidoc && babel-node index.js",
  "clusters": "babel-node clusters.js",
  "test": "NODE_ENV=test mocha test/**/*.js",
  "apidoc": "apidoc -i routes/ -o public/apidoc"
}
```

Agora, execute o comando `npm run clusters`. Dessa vez, a aplicação vai rodar de forma distribuída e, para comprovar que deu certo, você verá no terminal a mensagem `"NTask API -`

porta 3000".

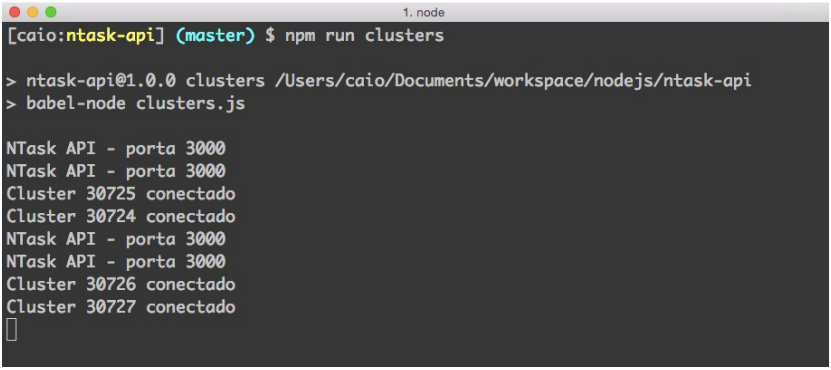


Figura 11.2: Rodando Node.js em clusters

Basicamente, carregamos o módulo `cluster` e, primeiro, verificamos se ele é o *cluster master* via função `cluster.isMaster`. Caso ele seja, rodamos um loop cujas iterações são baseadas no total de núcleos de processamento (*CPUs*) que ocorrem por meio do trecho `CPUS.forEach()`, que retorna o total de núcleos do servidor. Em cada iteração, rodamos o `cluster.fork()` que, na prática, instancia um processo filho *cluster slave*.

Quando nasce um novo processo (neste caso, um processo filho), consequentemente ele não cai na condicional `if(cluster.isMaster)`. Com isso, é iniciado o servidor da aplicação via `require("./index.js")` para este processo filho.

Também foram incluídos alguns eventos que são emitidos pelo *cluster master*. No código anterior, usamos apenas os principais eventos:

- `listening` : acontece quando um *cluster* está escutando

uma porta do servidor. Neste caso, a nossa aplicação está escutando **a porta 3000**;

- `disconnect` : executa seu callback quando um *cluster* se desconecta da rede;
- `exit` : ocorre quando um processo filho é fechado no sistema operacional.

> **Desenvolvimento em clusters**
>
> Muito pode ser explorado no desenvolvimento de *clusters* no Node.js. Aqui, apenas aplicamos o essencial para manter nossa aplicação rodando em paralelo. Mas, caso tenha a necessidade de implementar mais detalhes que explorem ao máximo os *clusters*, recomendo que leia a documentação (https://nodejs.org/api/cluster.html), para ficar por dentro de todos os eventos e funções deste módulo.

Para finalizar e deixar automatizado o comando `npm start` que inicia o servidor para ele rodar em modo *cluster*, atualize em seu `package.json` o atributo `scripts` de acordo com o código a seguir:

```
"scripts": {
  "start": "npm run apidoc && npm run clusters",
  "clusters": "babel-node clusters.js",
  "test": "NODE_ENV=test mocha test/**/*.js",
  "apidoc": "apidoc -i routes/ -o public/apidoc"
}
```

Pronto! Agora você pode levantar uma rede de *clusters* de sua aplicação pelo comando `npm start` .

11.5 COMPACTANDO REQUISIÇÕES COM GZIP

Para tornar as requisições mais leves, para consequentemente elas carregarem mais rápido, vamos habilitar mais um middleware em nossa aplicação que será responsável por compactar as respostas JSON e também todos os arquivos estáticos da documentação da API para o formato GZIP – um formato compatível com vários browsers. Vamos fazer essa simples, porém importante, alteração apenas usando o módulo `compression`. Instale-o via comando:

```
npm install compression@1.6.1 --save
```

Com ele já instalado, será necessário agora apenas incluir sua função como middleware no Express. Edite o `libs/middlewares.js` da seguinte maneira:

```
import bodyParser from "body-parser";
import express from "express";
import morgan from "morgan";
import cors from "cors";
import compression from "compression";
import logger from "./logger.js";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(morgan("common", {
    stream: {
      write: (message) => {
        logger.info(message);
      }
    }
  }));
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
```

```
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(compression());
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

Para testar essa compactação, basta reiniciar o servidor e, em seguida, acessar o endereço da documentação da API (afinal, lá existe muito arquivo estático que será compactado para GZIP): http://localhost:3000/apidoc.

Para visualizar em detalhes, abra o console do browser (Firefox e Google Chrome tem um ótimo console client-side) e acesse o menu **Redes**. Lá você verá o tamanho transferido *versus* o tamanho do arquivo, semelhante a esta figura:

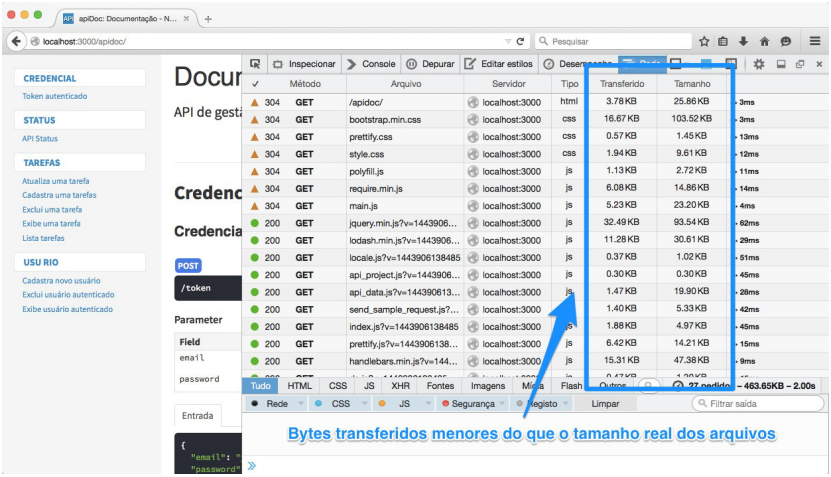


Figura 11.3: Compactação GZIP nas requisições

11.6 CONFIGURANDO SSL PARA USAR HTTPS

Hoje em dia, é mais que obrigação desenvolver uma aplicação segura, que forneça uma conexão segura entre cliente e servidor. Para isso, muitas aplicações compram e usam certificados de segurança para garantir uma conexão SSL (*Secure Sockets Layer*) por meio do uso do protocolo HTTPS.

Para implementar uma conexão com protocolo HTTPS em nossa aplicação, é necessário comprar um certificado digital para uso em ambiente de produção. No nosso caso, vamos trabalhar com um certificado fictício, não válido para uso em produção, e sim somente para fins didáticos. Para criar um certificado simples, você pode acessar o site: http://www.selfsignedcertificate.com.

Informe o domínio `ntask` da aplicação e clique em **Generate**. Uma nova tela vai aparecer com os dois arquivos de extensão `.key` e `.cert`. Faça download desses dois arquivos e mande-os para pasta raiz do nosso projeto.

Agora vamos utilizar o módulo nativo `https` para permitir que nosso servidor inicie pelo protocolo HTTPS. Para isso, vamos substituir a função `app.listen()` pela função `https.createServer(credentials, app).listen()` em nosso arquivo de inicialização da API. Para implementar essa funcionalidade, edite o `libs/boot.js`:

```
import https from "https";
import fs from "fs";

module.exports = app => {
  if (process.env.NODE_ENV !== "test") {
    const credentials = {
      key: fs.readFileSync("ntask.key", "utf8"),
      cert: fs.readFileSync("ntask.cert", "utf8")
```

```
    }
    app.db.sequelize.sync().done(() => {
      https.createServer(credentials, app)
        .listen(app.get("port"), () => {
          console.log(`NTask API - porta ${app.get("port")}`);
        });
    });
  }
};
```

Parabéns! Agora sua aplicação estará rodando em um protocolo mais seguro, garantindo que os dados não sejam interceptados. Vale lembrar que, para um projeto em produção, é preciso a compra de um certificado digital, jamais utilize esse certificado simples!

Para testar, basta reiniciar sua aplicação e acessar o endereço: https://localhost:3000/.

11.7 BLINDANDO A API COM HELMET

Finalizando o desenvolvimento de nossa API, vamos agora incluir um módulo muito importante, que é um middleware de segurança que trata vários tipos de ataques no protocolo HTTP. Esse módulo se chama `helmet`, e ele é um conjunto de 9 middlewares internos que tratam as seguintes configurações do HTTP:

- Configura o *Content Security Policy*;
- Remove o header `X-Powered-By` que informa o nome e versão do servidor;
- Configura regras para *HTTP Public Key Pinning*;
- Configura regras para *HTTP Strict Transport Security*;
- Trata o header `X-Download-Options` para *IE8+*;

- Desabilita *client-side caching*;
- Previne ataques do tipo *sniffing* no *Mime Type* do cliente;
- Previne ataques do tipo *ClickJacking*;
- Protege contra ataques do tipo XSS (*Cross-Site Scripting*).

Em resumo, mesmo que você não entenda muito sobre segurança, utilize-o, pois, além de ter uma simples interface, ele vai blindar sua aplicação web contra diversos tipos de ataques sobre o protocolo HTTP. Para instalá-lo, rode o comando:

```
npm install helmet@1.1.0 --save
```

Para garantir total segurança em nossa API, vamos usar todos os 9 middlewares do `helmet`, que é facilmente incluído via função `app.use(helmet())`. Então, edite o código `libs/middlewares.js` com a seguinte implementação:

```
import bodyParser from "body-parser";
import express from "express";
import morgan from "morgan";
import cors from "cors";
import compression from "compression";
import helmet from "helmet";
import logger from "./logger.js";

module.exports = app => {
  app.set("port", 3000);
  app.set("json spaces", 4);
  app.use(morgan("common", {
    stream: {
      write: (message) => {
        logger.info(message);
      }
    }
  }));
  app.use(helmet());
  app.use(cors({
    origin: ["http://localhost:3001"],
    methods: ["GET", "POST", "PUT", "DELETE"],
```

```
    allowedHeaders: ["Content-Type", "Authorization"]
  }));
  app.use(compression());
  app.use(bodyParser.json());
  app.use(app.auth.initialize());
  app.use((req, res, next) => {
    delete req.body.id;
    next();
  });
  app.use(express.static("public"));
};
```

Agora, reinicie sua aplicação e acesse pelo browser o endereço: http://localhost:3000/.

Abra o console do browser e, no menu **Redes**, visualize em detalhes os dados requisição da `GET /`. Lá você verá novos itens incluídos no cabeçalho de resposta, algo semelhante a esta figura:

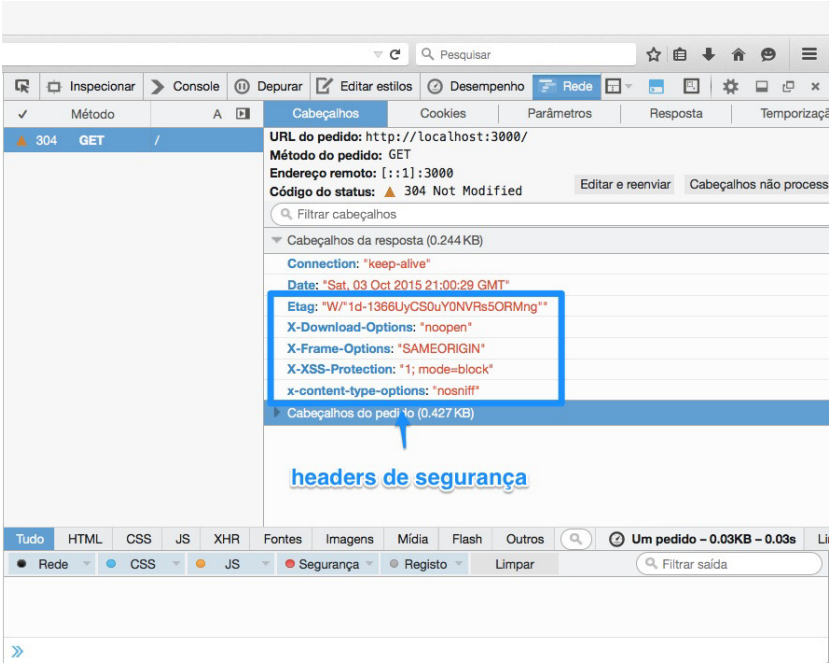


Figura 11.4: Headers de segurança

Conclusão

Congrats! Acabamos de finalizar o desenvolvimento completo de nossa API! Você pode usar esse projeto como base para seus futuros projetos de API Node.js, pois foi desenvolvida, na prática, uma API documentada que adota os principais padrões RESTful, possui testes em cima dos endpoints, persiste dados em banco de dados do tipo SQL via módulo Sequelize e, o mais importante, segue boas práticas de performance e segurança para rodar em ambiente de produção.

Mas calma! O livro ainda não acabou! Nos próximos capítulos, criaremos uma aplicação web que vai consumir dados da API. Ela

será uma simples SPA (*Single Page Application*), e será desenvolvida utilizando apenas o mais puro do JavaScript ES6, por meio do uso dos módulos `browserify e babel` no front-end.

Capítulo 12

CONSTRUINDO UMA APLICAÇÃO CLIENTE — PARTE 1

Depois de uma longa leitura sobre como construir um back-end de API RESTful utilizando Node.js e algumas boas práticas de desenvolvimento com a linguagem JavaScript EcmaScript 6, vamos criar, a partir deste capítulo, um novo projeto. Dessa vez, um projeto front-end usando o melhor do JavaScript EcmaScript 6!

Este livro apresentou 80% de conteúdo sobre desenvolvimento back-end, mas somente agora, neste capítulo, focaremos nos 20% de conteúdo front-end. Afinal, temos uma API, porém ela ainda não possui uma aplicação cliente, e os usuários somente interagem com aplicações clientes.

Por este motivo, nestes últimos capítulos, construiremos uma aplicação SPA (*Single Page Application*), utilizando apenas boas práticas de JavaScript puro. Isso mesmo! Apenas JavaScript ES6!

Não será usado nenhum framework de front-end (Angular, Backbone, Ember, React etc.), e também não será utilizado jQuery para manipulação do DOM (*Document Object Model*) do HTML. Apenas o melhor do Vanilla JavaScript!

12.1 SETUP DO AMBIENTE DA APLICAÇÃO

A nossa aplicação cliente será construída utilizando boas práticas de Orientação a Objetos (OO) do ES6, e Browserify para usar no front-end alguns módulos do NPM. Também vamos automatizar algumas tarefas de *build* da aplicação utilizando apenas comando *alias* do NPM, que é algo que foi usado bastante na construção da API.

Para começar essa brincadeira, vamos abrir o terminal em uma pasta qualquer de workspace de sua preferência. **Não** pode ser no mesmo diretório da API, pois esse será um novo projeto que vamos construir do zero.

Para iniciar este novo projeto, que será chamado de `ntask-web`, vamos rodar os seguintes comandos:

```
mkdir ntask-web
cd ntask-web
npm init
```

Com o comando `npm init`, vamos responder as seguintes perguntas:

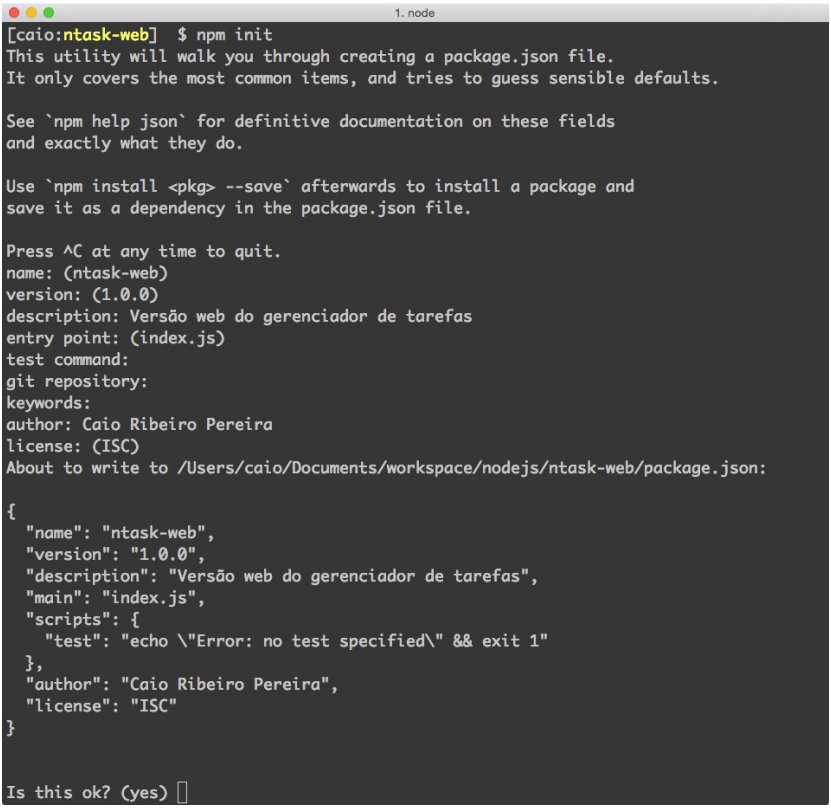

1. node
[caio:ntask-web] $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
See `npm help json` for definitive documentation on these fields
and exactly what they do.
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.
Press ^C at any time to quit.
name: (ntask-web)
version: (1.0.0)
description: Versão web do gerenciador de tarefas
entry point: (index.js)
test command:
git repository:
keywords:
author: Caio Ribeiro Pereira
license: (ISC)
About to write to /Users/caio/Documents/workspace/nodejs/ntask-web/package.json:
{
"name": "ntask-web",
"version": "1.0.0",
"description": "Versão web do gerenciador de tarefas",
"main": "index.js",
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1"
},
"author": "Caio Ribeiro Pereira",
"license": "ISC"
}
Is this ok? (yes)

Figura 12.1: Descrição do package.json do NTask Web

Após a execução do `npm init`, surgiu o arquivo `package.json` do nosso novo projeto. Crie na raiz do projeto os seguintes diretórios pelos comandos:

```
mkdir -p public/{css,fonts,js}
mkdir -p src/{components,templates}
```

No final, teremos a seguinte estrutura de diretórios:

- `public` : pasta para arquivos estáticos;
- `public/css` : diretório de CSS (vamos usar o CSS do

Ionic);

- `public/fonts` : diretório de fontes (vamos usar os fonticons do Ionic);
- `public/js` : diretório de JavaScript, aqui terá a versão final (compilada e minificada) do código JavaScript da nossa aplicação cliente;
- `src` : pasta com códigos JavaScript separado em módulos;
- `src/components` : pasta com códigos JavaScript de regras de negócio de cada página da aplicação;
- `src/templates` : pasta com códigos de templates (páginas da aplicação), que são Strings representando pedaços de HTML concatenados com dados de objetos que serão enviados pela API.

Agora, vamos instalar todos os módulos que serão utilizados em nossa aplicação cliente. Usaremos os seguintes módulos:

- `http-server` : CLI de servidor HTTP para arquivos estáticos (afinal, nossa aplicação cliente será construída apenas com HTML, CSS e JavaScript).
- `browserify` : um compilador JavaScript que permite utilizar módulos do NPM que são construídos com código JavaScript isomórfico (são códigos que funcionam tanto no back-end como no front-end), assim como também permite carregar códigos JavaScript no padrão CommonJS, o mesmo padrão do Node.js.
- `babelify` : um plugin para o browserify, baseado no Babel, para compilar códigos EcmaScript 6 no front-end.
- `babel-preset-es2015` : plugin babel para reconhecer códigos ES6.
- `tiny-emitter` : um módulo pequeno que permite

implementar e trabalhar de forma orientada a eventos.

- `browser-request` : é uma versão do módulo `request` focado para browsers, ele é *cross-browser* (compatível com os principais browsers) e abstrai toda complexidade de realizar uma requisição AJAX.

Basicamente, vamos construir um cliente web usando apenas esses módulos. Então, instale-os com os comandos:

```
npm install http-server@0.9.0 browserify@13.0.0 --save
npm install babelify@7.2.0 babel-preset-es2015@6.5.0 --save
npm install tiny-emitter@1.0.2 browser-request@0.3.3 --save
```

Após essa instalação, vamos modificar o `package.json` , removendo os atributos `main` , `script.test` e `license` , e adicionando todos os comandos *alias* que serão necessários para fazer um build do projeto front-end.

Basicamente, vamos:

- Criar *alias* para compilar e concatenar todos códigos da pasta `src` via browserify com babelify por meio do comando `npm run build` ;
- Iniciar o servidor na porta `3001` através do `npm run server` ;
- Criar o comando `npm start` , que é a execução dos comandos `npm run build` e `npm run server` .

Para aplicar essas alterações, edite o `package.json` para que ele fique exatamente igual a este:

```
{
  "name": "ntask-web",
  "version": "1.0.0",
  "description": "Versão web do gerenciador de tarefas",
```

```
  "scripts": {
    "start": "npm run build && npm run server",
    "server": "http-server public -p 3001",
    "build": "browserify src -t babelify -o public/js/app.js"
  },
  "author": "Caio Ribeiro Pereira",
  "dependencies": {
    "babel-preset-es2015": "^6.5.0",
    "babelify": "^7.2.0",
    "browser-request": "^0.3.3",
    "browserify": "^13.0.0",
    "http-server": "^0.9.0",
    "tiny-emitter": "^1.0.2"
  }
}
```

Assim como foi configurado no projeto `ntask-api`, também teremos que neste projeto incluir o arquivo `.babelrc` para que o módulo `babelify` utilize o plugin `babel-preset-es2015` para reconhecimento de códigos ES6 no projeto. Para isso crie o arquivo `.babelrc` na raíz do projeto:

```
{
  "presets": ["es2015"]
}
```

Após esse setup do ambiente da aplicação, incluiremos alguns arquivos estáticos que serão responsáveis pela estilização do layout e do conteúdo inicial da homepage do nosso projeto. Para não perder tempo, vamos utilizar uma estilização de CSS pronta, do framework Ionic, um framework muito legal que possui diversos componentes mobile para construção de aplicações web responsiva.

Não vamos usar o framework completo do Ionic, afinal, ele possui uma forte dependência do framework Angular. Vamos apenas incluir seu CSS e pacote de ícones. Para isso, recomendo que você faça o download dos arquivos que listarei a seguir e, em

seguida, envie os arquivos de CSS para o diretório `public/css` , e os arquivos de fontes para `public/fonts` :

- **CSS do Ionic:**

  http://code.ionicframework.com/1.0.0/css/ionic.min.css

- **CSS do Ionic icons:**

  http://code.ionicframework.com/ionicons/2.0.0/css/ionicons.min.css

- **Fontes do Ionic Icons:**

  - http://code.ionicframework.com/1.0.0/fonts/ionicons.eot
  - http://code.ionicframework.com/1.0.0/fonts/ionicons.svg
  - http://code.ionicframework.com/1.0.0/fonts/ionicons.ttf
  - http://code.ionicframework.com/1.0.0/fonts/ionicons.woff

Assim, criaremos a página inicial e código JavaScript para testarmos o comando `npm start` que inicializa a aplicação. O HTML principal será responsável por carregar a estilização CSS do Ionic, o JavaScript principal das interações da aplicação, e também terá o mínimo de tags HTML para montar a estrutura do layout. Para entender essa implementação, crie o arquivo `public/index.html` da seguinte maneira:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
```

```
    <title>NTask - Gerenciador de tarefas</title>
    <meta name="viewport"
                content="width=device-width,initial-scale=1">
    <link rel="stylesheet" href="css/ionic.min.css">
    <link rel="stylesheet" href="css/ionicons.min.css">
    <script src="js/app.js" async defer></script>
  </head>
  <body>
    <header class="bar bar-header bar-calm">
      <h1 class="title">NTask</h1>
    </header>
    <div class="scroll-content ionic-scroll">
      <div class="content overflow-scroll has-header">
        <main></main>
        <footer></footer>
      </div>
    </div>
  </body>
</html>
```

Perceba que existem duas tags vazias: a `<main></main>` e a `<footer></footer>`. Todas as regras de interação da aplicação serão criadas para manipular essas tags de forma dinâmica por meio dos futuros códigos JavaScript que vamos escrever em breve.

Para finalizar essa seção inicial, crie o `src/index.js` com um código que, por enquanto, exibirá uma simples mensagem de `Bem-vindo!` no browser quando carregar a página. Isso será modificado em breve, afinal, vamos criá-lo agora apenas para testar se o ambiente da aplicação está funcionando corretamente.

```
window.onload = () => {
  alert("Bem-vindo!");
};
```

Pronto. Agora já temos um ambiente simples, porém funcional, para construirmos o front-end da aplicação NTask Web. Para testá-lo, execute o comando `npm start` e, em seguida, acesse o endereço: http://localhost:3001.

Se tudo estiver funcionando direito, você terá o seguinte resultado:

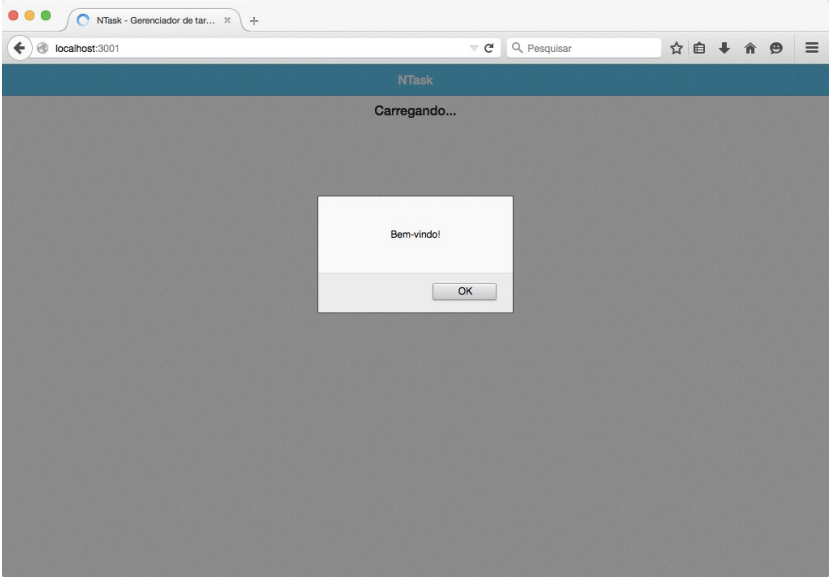


Figura 12.2: Primeira tela do NTask Web

12.2 CRIANDO TEMPLATES DE SIGNIN E SIGNUP

Nesta seção, vamos criar todos os templates que serão utilizados em nossa aplicação cliente. Os templates são basicamente pedaços de HTML, manipulados via JavaScript, e são largamente utilizados em sistemas do tipo SPA (*Single Page Application*, ou seja, aplicações de uma única página). Afinal, a filosofia de uma SPA é carregar todos os arquivos estáticos uma única vez (HTML, CSS, JavaScript, imagens etc.), para que somente os dados sejam requisitados com frequência do servidor.

Toda responsabilidade de transição de telas (transição de templates) e concatenação de dados do servidor com as telas se tornam tarefas da aplicação cliente, fazendo com que o servidor trafegue apenas dados, e o cliente trate de pegar os dados para montar as devidas telas para o usuário final interagir na aplicação.

Nossa aplicação é um simples gerenciador de tarefas, que possui uma API REST com endpoints para criar; atualizar; excluir e listar tarefas; e cadastrar, consultar e excluir um usuário. Os templates serão baseados nessas funcionalidades que a API fornece atualmente. Então, não há nada a inventar, e sim botar a mão na massa baseado nos endpoints da API.

Para começar, vamos construir o template que será a tela de *sign in* e *sign up* da aplicação. Graças à funcionalidade de Template String do EcmaScript 6, se tornou possível criar strings com concatenação de dados de forma mais elegante através da sintaxe `Olá ${nome}!`. Com isso, não será necessário usar nenhum framework de template engine, pois podemos facilmente criar os templates utilizando apenas uma função que retorna uma string de HTML concatenada com dados.

Para entender melhor essa implementação, vamos começar criando a tela inicial de sign in que, por meio da função `render()`, retornará uma String de HTML, ou seja, o template da tela sign in. Crie o arquivo `src/templates/signin.js` com o seguinte código:

```
exports.render = () => {
  return `<form>
    <div class="list">
      <label class="item item-input item-stacked-label">
        <span class="input-label">Email</span>
        <input type="text" data-email>
```

```
      </label>
      <label class="item item-input item-stacked-label">
        <span class="input-label">Senha</span>
        <input type="password" data-password>
      </label>
    </div>
    <div class="padding">
      <button class="button button-positive button-block">
        <i class="ion-home"></i> Entrar
      </button>
    </div>
  </form>
  <div class="padding">
    <button class="button button-block" data-signup>
      <i class="ion-person-add"></i> Cadastrar
    </button>
  </div>`;
};
```

Agora, para completar o fluxo, vamos criar também o template da tela de sign up (cadastro de usuário). Crie o arquivo `src/templates/signup.js` da seguinte maneira:

```
exports.render = () => {
  return `<form>
    <div class="list">
      <label class="item item-input item-stacked-label">
        <span class="input-label">Nome</span>
        <input type="text" data-name>
      </label>
      <label class="item item-input item-stacked-label">
        <span class="input-label">Email</span>
        <input type="text" data-email>
      </label>
      <label class="item item-input item-stacked-label">
        <span class="input-label">Senha</span>
        <input type="password" data-password>
      </label>
    </div>
    <div class="padding">
      <button class="button button-positive button-block">
        <i class="ion-thumbsup"></i> Cadastrar
      </button>
```

```
    </div>
  </form>`;
};
```

Pronto! Já temos duas telas importantes da aplicação. Agora, só falta criarmos os códigos de interação dessas páginas. Eles serão responsáveis por renderizar esses templates e, principalmente, programar os eventos de cada componente do template, para que eles realizem suas devidas comunicações com a API.

12.3 IMPLEMENTANDO OS COMPONENTES DE SIGN IN E SIGN UP

Os códigos de interação dos templates serão colocados na pasta `src/components`, mas antes de criá-los, vamos explorar duas novas funcionalidades do JavaScript ES6: classes e herança. Para deixar mais semântico e organizado, criaremos uma classe pai que terá apenas dois atributos importantes que todas as classes de componentes vão herdar: `this.URL` (aqui terá o endereço URL da API) e `this.request` (aqui será carregado o módulo `browser-request`).

Outro detalhe dessa classe pai é que ela vai herdar todas as funcionalidades do módulo `tiny-emitter` (via linha `class NTask extends TinyEmitter`), que repassará essas funcionalidades também para suas classes filhas, permitindo que elas emitam e escutem os eventos. Para entender melhor essa classe, crie o arquivo `src/ntask.js`:

```
import TinyEmitter from "tiny-emitter";
import Request from "browser-request";

class NTask extends TinyEmitter {
  constructor() {
```

```
    super();
    this.request = Request;
    this.URL = "https://localhost:3000";
  }
}
module.exports = NTask;
```

Agora que temos a classe pai `NTask`, torna-se possível construir as classes de componentes que, não só terão suas funcionalidades específicas, como também terão atributos e funções genéricas herdadas da classe pai. Ou seja, os componentes, em vez de duplicar ou triplicar códigos, eles vão reaproveitar códigos.

Vamos criar nosso primeiro componente, que será a tela de sign in. O padrão das classes dos componentes de nossa aplicação terá sempre um construtor recebendo um objeto `body` (`constructor(body)`). Esse `body` será basicamente o objeto DOM da tag `<main>`, ou tag `<footer>` da página principal, tudo vai depender do que será esse componente.

Todos os componentes vão herdar da classe pai `NTask`, logo, é obrigatório a execução da função `super()` no início do construtor da classe filha. Outro padrão que vamos adotar para organizar os nossos componentes será o uso dos métodos: `render()` (responsável por renderizar um módulo de template) e `addEventListener()` (responsável por fazer um escuta e tratamento dos eventos de botões, links ou formulários, ou seja, componentes do HTML dos templates).

Neste caso, teremos dois eventos encapsulados nos métodos: `formSubmit()` (responsável por fazer uma requisição de autenticação de usuário na API) e `signupClick()` (responsável por mostrar o template da tela de cadastro, a tela de sign up).

Toda resposta final de um componente do template deve emitir um evento pela função `this.emit("nome-do-evento")`, pois vamos criar em breve uma classe observadora de eventos que será responsável por delegar tarefas entre os demais componentes, de acordo com os eventos emitidos. Um bom exemplo de tarefas que será largamente usada é a de transição entre templates, que ocorre quando um usuário clica no botão de um template e, no evento de click, a classe observadora delega a tarefa para um novo template renderizar sua tela.

Para entender melhor essas regras, crie o arquivo `src/components/signin.js` com os seguintes códigos:

```
import NTask from "../ntask.js";
import Template from "../templates/signin.js";

class Signin extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.body.innerHTML = Template.render();
    this.body.querySelector("[data-email]").focus();
    this.addEventListener();
  }
  addEventListener() {
    this.formSubmit();
    this.signupClick();
  }
  formSubmit() {
    const form = this.body.querySelector("form");
    form.addEventListener("submit", (e) => {
      e.preventDefault();
      const email = e.target.querySelector("[data-email]");
      const password = e.target.querySelector("[data-password]");
      const opts = {
        method: "POST",
        url: `${this.URL}/token`,
        json: true,
```

```
        body: {
          email: email.value,
          password: password.value
        }
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 401) {
          this.emit("error", err);
        } else {
          this.emit("signin", data.token);
        }
      });
    });
  }
  signupClick() {
    const signup = this.body.querySelector("[data-signup]");
    signup.addEventListener("click", (e) => {
      e.preventDefault();
      this.emit("signup");
    });
  }
}
module.exports = Signin;
```

Também criaremos a classe de componentes do `Signup`. Ela seguirá o mesmo padrão da classe `Signin`. Para ver como que ela deve ficar, crie o `src/components/signup.js` da seguinte forma:

```
import NTask from "../ntask.js";
import Template from "../templates/signup.js";

class Signup extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.body.innerHTML = Template.render();
    this.body.querySelector("[data-name]").focus();
    this.addEventListener();
  }
  addEventListener() {
    this.formSubmit();
```

```
  }
  formSubmit() {
    const form = this.body.querySelector("form");
    form.addEventListener("submit", (e) => {
      e.preventDefault();
      const name = e.target.querySelector("[data-name]");
      const email = e.target.querySelector("[data-email]");
      const password = e.target.querySelector("[data-password]");
      const opts = {
        method: "POST",
        url: `${this.URL}/users`,
        json: true,
        body: {
          name: name.value,
          email: email.value,
          password: password.value
        }
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 412) {
          this.emit("error", err);
        } else {
          this.emit("signup", data);
        }
      });
    });
  }
}
module.exports = Signup;
```

Para finalizar esse fluxo inicial da aplicação, falta apenas criar a classe observadora e, em seguida, carregá-la dentro do `src/index.js`, que é o código responsável por iniciar toda interação dos componentes. A classe observadora se chamará `App`. Seu construtor realizará a instância de todos os objetos componentes, e ela terá dois métodos principais: o `init()` (responsável por iniciar os componentes) e `addEventListener()` (responsável por escutar e tratar os eventos dos componentes).

Para que nossa aplicação apresente as primeiras interações de

tela de sign in e sign up, crie o `src/app.js`, seguindo esse código inicial:

```
import Signin from "./components/signin.js";
import Signup from "./components/signup.js";

class App {
  constructor(body) {
    this.signin = new Signin(body);
    this.signup = new Signup(body);
  }
  init() {
    this.signin.render();
    this.addEventListener();
  }
  addEventListener() {
    this.signinEvents();
    this.signupEvents();
  }
  signinEvents() {
    this.signin.on("error", () => alert("Erro de autenticação"));
    this.signin.on("signin", (token) => {
      localStorage.setItem("token", `JWT ${token}`);
      alert("Você esta autenticado!");
    });
    this.signin.on("signup", () => this.signup.render());
  }
  signupEvents() {
    this.signup.on("error", () => alert("Erro no cadastro"));
    this.signup.on("signup", (user) => {
      alert(`${user.name} você foi cadastrado com sucesso!`);
      this.signin.render();
    });
  }
}
module.exports = App;
```

Basicamente foram criados os eventos de sucesso na autenticação, erro na autenticação, acesso à tela de cadastro, sucesso no cadastro e erro no cadastro. O método `init()` inicia renderizando a tela inicial, que é a tela de sign in e, em seguida, executa o método `addEventListener()`, para escutar todos os

eventos dos componentes que estão encapsulados nos métodos `signinEvents()` e `signupEvents()`.

Para finalizar, vamos editar o `src/index.js` para que, por meio do evento `window.onload()`, ele inicie a classe `App` para dar início a todo o fluxo interativo da aplicação. Edite-o da seguinte maneira:

```
import App from "./app.js"

window.onload = () => {
  const main = document.querySelector("main");
  new App(main).init();
};
```

Vamos testar? Se você seguiu passo a passo essas primeiras implementações, você terá um fluxo básico de sign in e sign up. Para rodar a aplicação, será necessário ter duas abas de terminal abertas: uma para iniciar o servidor da API através do comando `npm start`, e outra na pasta desse projeto cliente.

Execute também o comando `npm start`. Se tudo estiver rodando corretamente, você terá dois endereços disponíveis:

- Endereço da API: https://localhost:3000
- Endereço do cliente web: http://localhost:3001

Como estamos usando um certificado digital não válido para produção, é bem provável que seu browser bloqueie o acesso à API. Caso isso aconteça, basta acessar o endereço: https://localhost:3000/.

Depois, procure em seu browser como adicionar exceção ao acesso à nossa API. Veja na figura seguir, como adicionar exceção, por exemplo, no Mozilla Firefox:

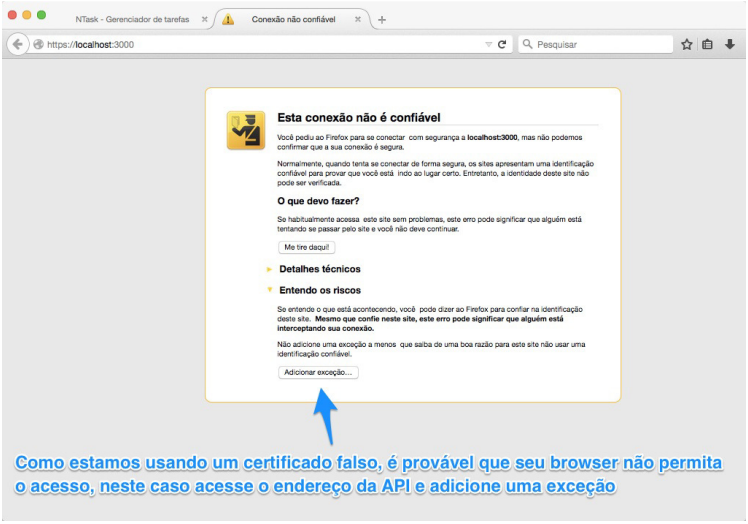

NTask - Gerenciador de tarefas
Conexão não confiável
https://localhost:3000
Pesquisar
Esta conexão não é confiável
Você pediu ao Firefox para se conectar com segurança a localhost:3000, mas não podemos confirmar que a sua conexão é segura.
Normalmente, quando tenta se conectar de forma segura, os sites apresentam uma identificação confiável para provar que você está indo ao lugar certo. Entretanto, a identidade deste site não pode ser verificada.
O que devo fazer?
Se habitualmente acessa este site sem problemas, este erro pode significar que alguém está tentando se passar pelo site e você não deve continuar.
Me tire daqui!
Detalhes técnicos
Entendo os riscos
Se entende o que está acontecendo, você pode dizer ao Firefox para confiar na identificação deste site. Mesmo que confie neste site, este erro pode significar que alguém está interceptando sua conexão.
Não adicione uma exceção a menos que saiba de uma boa razão para este site não usar uma identificação confiável.
Adicionar exceção...
Como estamos usando um certificado falso, é provável que seu browser não permita o acesso, neste caso acesse o endereço da API e adicione uma exceção

Figura 12.3: Adicionando exceção ao acesso à API

Agora que a API esta com acesso permitido pelo seu browser, acesse o endereço do cliente web: http://localhost:3001.

Se tudo estiver correto, você terá acesso às seguintes telas:

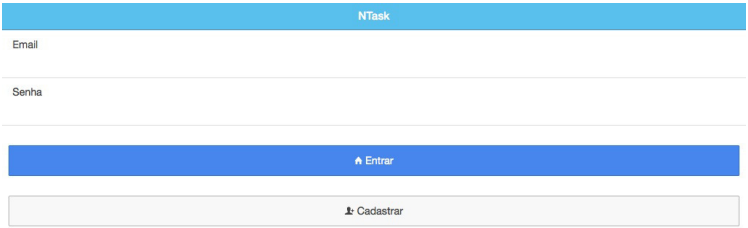


Figura 12.4: Tela inicial de Signin

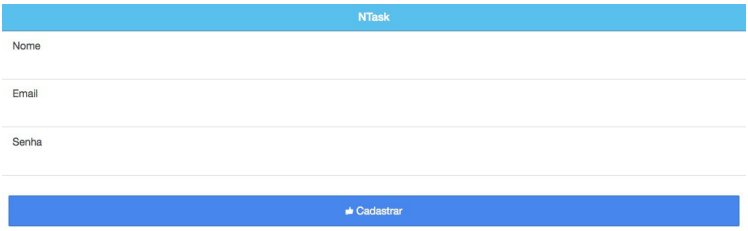


Figura 12.5: Tela de cadastro de usuário

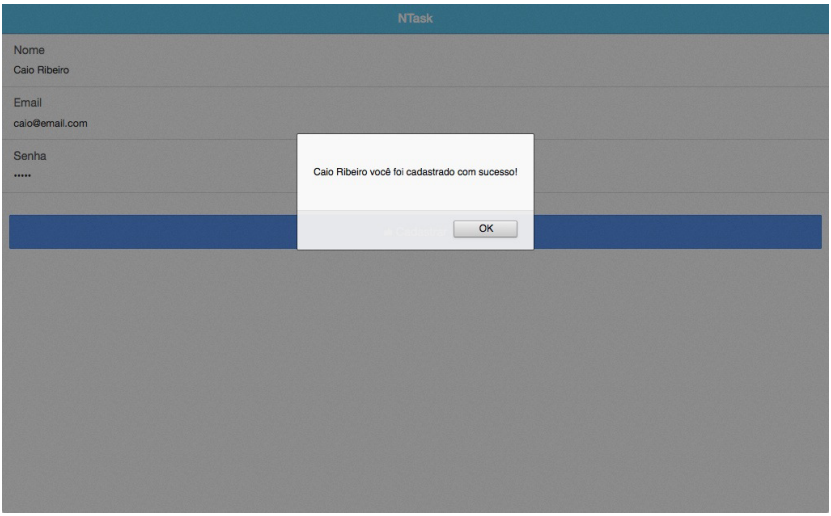


Figura 12.6: Cadastrando um novo usuário

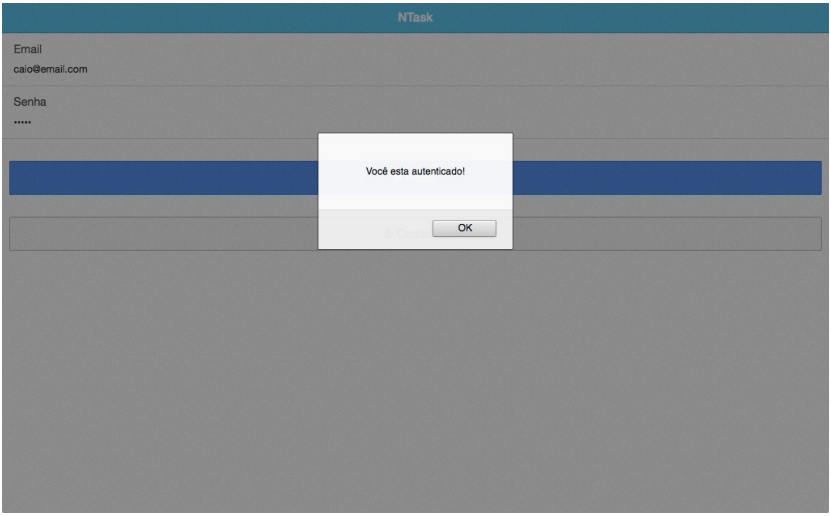


Figura 12.7: Logando com nova conta cadastrada

Conclusão

Nosso novo projeto está ganhando forma, e estamos cada vez mais próximos de construir um sistema funcional para o usuário final, tudo isso através da integração da aplicação cliente com a API que já foi construída capítulos atrás.

Neste capítulo, foi criado apenas o ambiente e algumas telas do projeto, suficiente para estruturar o front-end da aplicação NTask. Continue lendo, pois no próximo capítulo vamos aprofundar mais na implementação das telas finais.

Capítulo 13

CONSTRUINDO UMA APLICAÇÃO CLIENTE — PARTE 2

Dando continuidade à construção da aplicação cliente, até agora temos uma aplicação com layout, que apenas se conecta à API e permite autenticar um usuário para acessar a aplicação cliente. Neste capítulo, vamos construir as funcionalidades principais para gestão de tarefas do usuário autenticado.

13.1 TEMPLATES E COMPONENTES PARA CRUD DE TAREFAS

A construção do template de tarefas será um pouco complexa, porém terá um resultado final bem legal! Esse template terá de listar todas as tarefas existentes de um usuário, então sua função receberá como argumento uma lista de tarefas e, por meio da função `tasks.map()`, será criado um array de templates referente a cada tarefa.

No final da geração desse novo array, é executada a função `.join("")`, que será responsável por concatenar todos os seus itens em uma única string de template das tarefas. Para facilitar

essa manipulação e geração do template de tarefas, essa lógica será encapsulada através da função `renderTasks(tasks)`, pois o retorno desta será concatenado em uma string de template final, responsável por verificar se existem tarefas para exibir todas as tarefas. Caso contrário, mostrará uma mensagem de que não existe nenhuma tarefa.

Para entender melhor a implementação desse template, crie o arquivo `src/templates/tasks.js` da seguinte maneira:

```
const renderTasks = tasks => {
  return tasks.map(task => {
    let done = task.done ? "ios-checkmark" :
                                          "ios-circle-outline";
    return `<li class="item item-icon-left item-button-right">
      <i class="icon ion-${done}" data-done
        data-task-done="${task.done ? 'done' : ''}"
        data-task-id="${task.id}"></i>
      ${task.title}
      <button data-remove data-task-id="${task.id}"
        class="button button-assertive">
        <i class="ion-trash-a"></i>
      </button>
    </li>`;
  }).join("");
};
exports.render = tasks => {
  if (tasks && tasks.length) {
    return `<ul class="list">${renderTasks(tasks)}</ul>`;
  }
  return `<h4 class="text-center">Nenhuma tarefa ainda</h4>`;
};
```

É por meio dos atributos `data-task-done`, `data-task-id`, `data-done` e `data-remove`, que trabalharemos para criar o componente de manipulação de tarefas. Esses atributos serão usados para criar os eventos necessários, para permitir excluir uma tarefa (via método `taskRemoveClick()`) e/ou definir que tal

tarefa foi concluída (via método `taskDoneCheckbox()`). Essas lógicas de interação desse template serão escritas no `src/components/tasks.js` a seguir:

```
import NTask from "../ntask.js";
import Template from "../templates/tasks.js";

class Tasks extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.renderTaskList();
  }
  addEventListener() {
    this.taskDoneCheckbox();
    this.taskRemoveClick();
  }
  renderTaskList() {
    const opts = {
      method: "GET",
      url: `${this.URL}/tasks`,
      json: true,
      headers: {
        authorization: localStorage.getItem("token")
      }
    };
    this.request(opts, (err, resp, data) => {
      if (err) {
        this.emit("error", err);
      } else {
        this.body.innerHTML = Template.render(data);
        this.addEventListener();
      }
    });
  }
  taskDoneCheckbox() {
    const dones = this.body.querySelectorAll("[data-done]");
    for(let i = 0, max = dones.length; i < max; i++) {
      dones[i].addEventListener("click", (e) => {
        e.preventDefault();
        const id = e.target.getAttribute("data-task-id");
```

```
      const done = e.target.getAttribute("data-task-done");
      const opts = {
        method: "PUT",
        url: `${this.URL}/tasks/${id}`,
        headers: {
          authorization: localStorage.getItem("token"),
          "Content-Type": "application/json"
        },
        body: JSON.stringify({
          done: !done
        })
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 412) {
          this.emit("update-error", err);
        } else {
          this.emit("update");
        }
      });
    });
  }
}
taskRemoveClick() {
  const removes = this.body.querySelectorAll("[data-remove]");
  for(let i = 0, max = removes.length; i < max; i++) {
    removes[i].addEventListener("click", (e) => {
      e.preventDefault();
      if (confirm("Deseja excluir esta tarefa?")) {
        const id = e.target.getAttribute("data-task-id");
        const opts = {
          method: "DELETE",
          url: `${this.URL}/tasks/${id}`,
          headers: {
            authorization: localStorage.getItem("token")
          }
        };
        this.request(opts, (err, resp, data) => {
          if (err || resp.status === 412) {
            this.emit("remove-error", err);
          } else {
            this.emit("remove");
          }
        });
      }
    });
```

```
    }
  }
}
module.exports = Tasks;
```

Agora que temos um componente responsável por listar, atualizar e excluir uma tarefa, vamos implementar o template e componente responsável por adicionar uma nova. Isso será mais fácil de criar, pois será um template com um simples formulário para cadastrar uma tarefa e, no final, redirecionará para uma lista de tarefas. Para criá-lo, crie o arquivo `src/templates/taskForm.js`:

```
exports.render = () => {
  return `<form>
    <div class="list">
      <label class="item item-input item-stacked-label">
        <span class="input-label">Tarefa</span>
        <input type="text" data-task>
      </label>
    </div>
    <div class="padding">
      <button class="button button-positive button-block">
        <i class="ion-compose"></i> Salvar
      </button>
    </div>
  </form>`;
};
```

Em seguida, crie o seu respectivo componente que terá apenas o evento de submissão do formulário encapsulado pela função `formSubmit()`. Para criá-lo, crie o arquivo `src/components/taskForm.js`:

```
import NTask from "../ntask.js";
import Template from "../templates/taskForm.js";

class TaskForm extends NTask {
  constructor(body) {
    super();
```

```
    this.body = body;
  }
  render() {
    this.body.innerHTML = Template.render();
    this.body.querySelector("[data-task]").focus();
    this.addEventListener();
  }
  addEventListener() {
    this.formSubmit();
  }
  formSubmit() {
    const form = this.body.querySelector("form");
    form.addEventListener("submit", (e) => {
      e.preventDefault();
      const task = e.target.querySelector("[data-task]");
      const opts = {
        method: "POST",
        url: `${this.URL}/tasks`,
        json: true,
        headers: {
          authorization: localStorage.getItem("token")
        },
        body: {
          title: task.value
        }
      };
      this.request(opts, (err, resp, data) => {
        if (err || resp.status === 412) {
          this.emit("error");
        } else {
          this.emit("submit");
        }
      });
    });
  }
}
module.exports = TaskForm;
```

13.2 COMPONENTES PARA TELA DE USUÁRIO LOGADO

Para terminar a criação de telas da nossa aplicação, vamos

construir a última tela que exibirá dados do usuário logado e um botão para ele cancelar a conta na aplicação. Essa tela também terá um componente fácil de implementar, pois precisará apenas tratar o evento do botão de cancelamento de conta. Para criá-la, primeiro crie o `src/templates/user.js` :

```
exports.render = user => {
  return `<div class="list">
    <label class="item item-input item-stacked-label">
      <span class="input-label">Nome</span>
      <small class="dark">${user.name}</small>
    </label>
    <label class="item item-input item-stacked-label">
      <span class="input-label">Email</span>
      <small class="dark">${user.email}</small>
    </label>
  </div>
  <div class="padding">
    <button data-remove-account
      class="button button-assertive button-block">
      <i class="ion-trash-a"></i> Excluir conta
    </button>
  </div>`;
};
```

Agora que temos o template da tela de usuário, crie seu respectivo componente pelo arquivo `src/components/user.js` , seguindo essa implementação de código:

```
import NTask from "../ntask.js";
import Template from "../templates/user.js";

class User extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render() {
    this.renderUserData();
  }
  addEventListener() {
```

```
    this.userCancelClick();
  }
  renderUserData() {
    const opts = {
      method: "GET",
      url: `${this.URL}/user`,
      json: true,
      headers: {
        authorization: localStorage.getItem("token")
      }
    };
    this.request(opts, (err, resp, data) => {
      if (err || resp.status === 412) {
        this.emit("error", err);
      } else {
        this.body.innerHTML = Template.render(data);
        this.addEventListener();
      }
    });
  }
  userCancelClick() {
    const button =
                this.body.querySelector("[data-remove-account]");
    button.addEventListener("click", (e) => {
      e.preventDefault();
      if (confirm("Tem certeza que deseja excluir sua conta?")) {
        const opts = {
          method: "DELETE",
          url: `${this.URL}/user`,
          headers: {
            authorization: localStorage.getItem("token")
          }
        };
        this.request(opts, (err, resp, data) => {
          if (err || resp.status === 412) {
            this.emit("remove-error", err);
          } else {
            this.emit("remove-account");
          }
        });
      }
    });
  }
}
module.exports = User;
```

13.3 CRIANDO COMPONENTE DE MENU DA APLICAÇÃO

Para deixar a aplicação mais bonita e interativa, vamos também criar em seu rodapé um menu para que o usuário acesse a lista de tarefas, o formulário para cadastro de nova tarefa ou a tela de dados do usuário. Para criar essa tela, primeiro criaremos o seu template, que inicialmente terá três botões: lista de tarefas, cadastro de tarefa e *logout* da aplicação. Crie o arquivo `src/templates/footer.js` da seguinte maneira:

```
exports.render = path => {
  let isTasks = path === "tasks" ? "active" : "";
  let isTaskForm = path === "taskForm" ? "active" : "";
  let isUser = path === "user" ? "active" : "";
  return `
    <div class="tabs-striped tabs-color-calm">
      <div class="tabs">
        <a data-path="tasks" class="tab-item ${isTasks}">
          <i class="icon ion-home"></i>
        </a>
        <a data-path="taskForm" class="tab-item ${isTaskForm}">
          <i class="icon ion-compose"></i>
        </a>
        <a data-path="user" class="tab-item ${isUser}">
          <i class="icon ion-person"></i>
        </a>
        <a data-logout class="tab-item">
          <i class="icon ion-android-exit"></i>
        </a>
      </div>
    </div>`;
};
```

Em seguida, crie seu respectivo componente, o `src/components/menu.js`:

```
import NTask from "../ntask.js";
```

```
import Template from "../templates/footer.js";

class Menu extends NTask {
  constructor(body) {
    super();
    this.body = body;
  }
  render(path) {
    this.body.innerHTML = Template.render(path);
    this.addEventListener();
  }
  clear() {
    this.body.innerHTML = "";
  }
  addEventListener() {
    this.pathsClick();
    this.logoutClick();
  }
  pathsClick() {
    const links = this.body.querySelectorAll("[data-path]");
    for(let i = 0, max = links.length; i < max; i++) {
      links[i].addEventListener("click", (e) => {
        e.preventDefault();
        const link = e.target.parentElement;
        const path = link.getAttribute("data-path");
        this.emit("click", path);
      });
    }
  }
  logoutClick() {
    const link = this.body.querySelector("[data-logout]");
    link.addEventListener("click", (e) => {
      e.preventDefault();
      this.emit("logout");
    })
  }
}
module.exports = Menu;
```

13.4 TRATANDO OS EVENTOS DOS COMPONENTES DAS TELAS

Nosso projeto possui todos os componentes necessários para construir uma aplicação de gestão de tarefas, tudo o que falta agora é juntar as peças do quebra-cabeça! A começar, vamos modificar o `src/index.js` para que ele manipule, não só a tag `<main>`, mas também a `<footer>`, pois essa nova tag será usada no menu da aplicação.

Edite o `src/index.js` aplicando essa simples modificação:

```
import App from "./app.js"

window.onload = () => {
  const main = document.querySelector("main");
  const footer = document.querySelector("footer");
  new App(main, footer).init();
};
```

Agora, para finalizar nosso projeto, temos de atualizar o objeto `App` para que ele seja responsável por carregar todos os componentes que foram criados e, principalmente, tratar os eventos de cada componente. Isso para garantir o fluxo correto de transição das telas, do menu e do tráfego de dados entre o `ntask-api` com `ntask-web`. Para isso, edite o `src/app.js` com o seguinte código:

```
import Tasks from "./components/tasks.js";
import TaskForm from "./components/taskForm.js";
import User from "./components/user.js";
import Signin from "./components/signin.js";
import Signup from "./components/signup.js";
import Menu from "./components/menu.js";

class App {
  constructor(body, footer) {
    this.signin = new Signin(body);
    this.signup = new Signup(body);
    this.tasks = new Tasks(body);
    this.taskForm = new TaskForm(body);
    this.user = new User(body);
```

```
    this.menu = new Menu(footer);
  }
  init() {
    this.signin.render();
    this.addEventListener();
  }
  addEventListener() {
    this.signinEvents();
    this.signupEvents();
    this.tasksEvents();
    this.taskFormEvents();
    this.userEvents();
    this.menuEvents();
  }
  signinEvents() {
    this.signin.on("error", () => alert("Erro de autenticação"));
    this.signin.on("signin", (token) => {
      localStorage.setItem("token", `JWT ${token}`);
      this.menu.render("tasks");
      this.tasks.render();
    });
    this.signin.on("signup", () => this.signup.render());
  }
  signupEvents(){
    this.signup.on("error", () => alert("Erro no cadastro"));
    this.signup.on("signup", (user) => {
      alert(`${user.name} você foi cadastrado com sucesso!`);
      this.signin.render();
    });
  }
  tasksEvents() {
    this.tasks.on("error", () =>
                              alert("Erro ao listar tarefas"));
    this.tasks.on("remove-error", () =>
                                  alert("Erro ao excluir"));
    this.tasks.on("update-error", () =>
                                alert("Erro ao atualizar"));
    this.tasks.on("remove", () => this.tasks.render());
    this.tasks.on("update", () => this.tasks.render());
  }
  taskFormEvents() {
    this.taskForm.on("error", () =>
                             alert("Erro ao cadastrar tarefa"));
    this.taskForm.on("submit", () => {
      this.menu.render("tasks");
```

```
      this.tasks.render();
    });
  }
  userEvents() {
    this.user.on("error", () => alert("Erro carregar usuário"));
    this.user.on("remove-error", () =>
                                alert("Erro ao excluir conta"));
    this.user.on("remove-account", () => {
      alert("Que pena! Sua conta foi excluída.");
      localStorage.clear();
      this.menu.clear();
      this.signin.render();
    });
  }
  menuEvents() {
    this.menu.on("click", (path) => {
      this.menu.render(path);
      this[path].render();
    });
    this.menu.on("logout", () => {
      localStorage.clear();
      this.menu.clear();
      this.signin.render();
    })
  }
}
module.exports = App;
```

Uffaa! Acabou! Terminamos de construir nossa aplicação cliente. Vamos testar? Basta apenas reiniciar a aplicação cliente e usá-la normalmente. A seguir, veja como ficaram as novas telas que criamos:

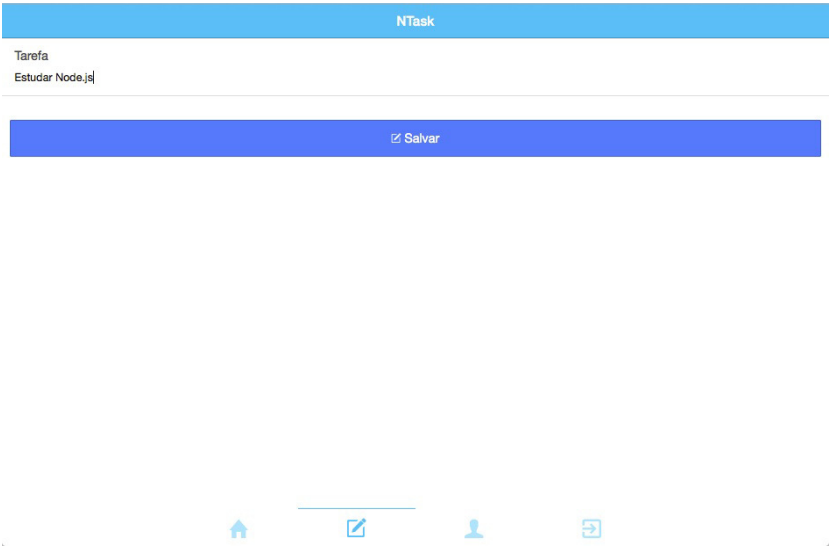


Figura 13.1: Cadastrando nova tarefa

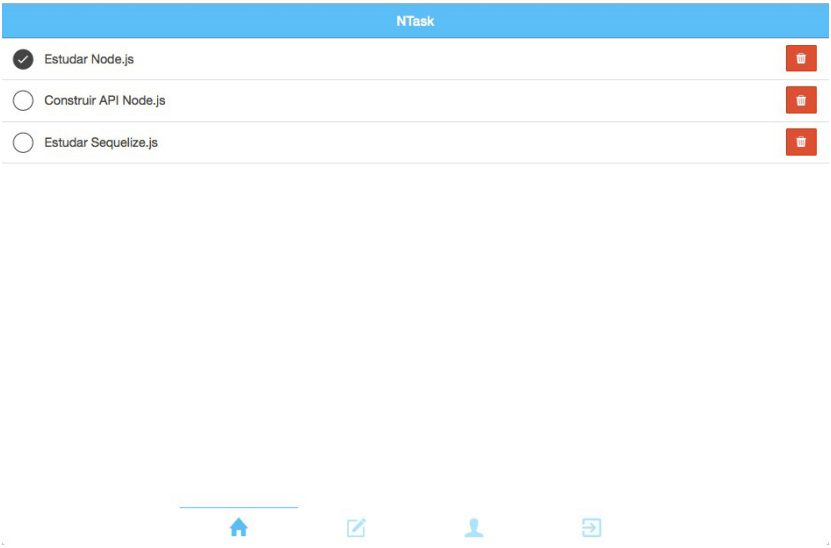


Figura 13.2: Listando e completando tarefas

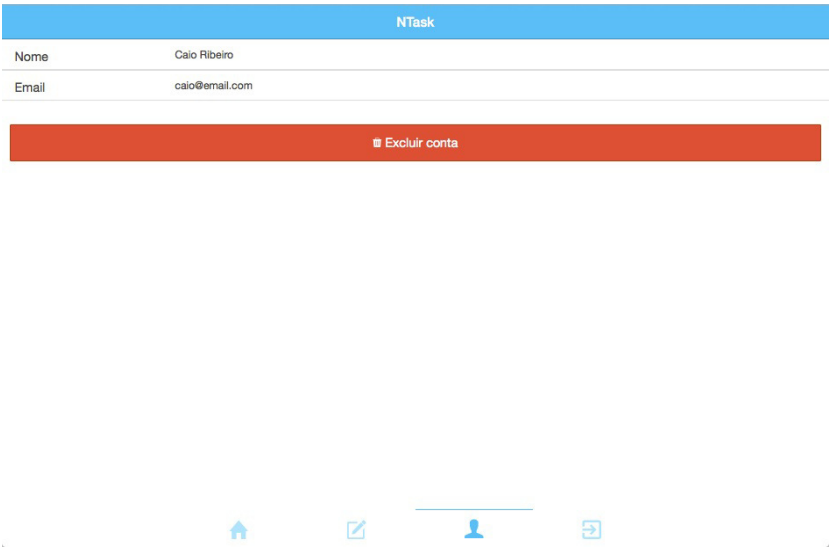


Figura 13.3: Tela de dados do usuário

Conclusão final

Parabéns! Se você chegou aqui com a aplicação rodando perfeitamente, então você finalizou este livro com sucesso. Espero que ele tenha ampliado seus conhecimentos técnicos com a plataforma Node.js e, principalmente, sobre como construir uma API RESTful, pois essa é a sua essência. Acredito ter passado os conhecimentos necessários para você, fiel leitor.

Vale lembrar que todo o código-fonte pode ser consultado em meu GitHub pessoal, e discutir sobre este livro no fórum da Casa do Código. Os links são:

- **NTask API:** https://github.com/caio-ribeiro-pereira/ntask-api
- **NTask Web:** https://github.com/caio-ribeiro-

pereira/ntask-web

- **Fórum da Casa do Código:** http://forum.casadocodigo.com.br

Muito obrigado por ler este livro!

Capítulo 14

REFERÊNCIAS BIBLIOGRÁFICAS

PEREIRA, Caio Ribeiro. *Aplicações web real-time com Node.js*. São Paulo: Casa do Código, 2013.

RAUCH, Guilhermo. *Smashing Node.js:* Javascript Everywhere. Wiley, 2012.

WILSON, Jim R. *Node.js the Right Way:* Practical, Server-Side JavaScript That Scales. The Pragmatic Bookshelf, 2013.

TEIXEIRA, Pedro. *Hands-on Node.js*. Leanpub, 2012.

MORAES, Willian Bruno. *Construindo aplicações com NodeJS*. Novatec, 2015.